UT700 LEAP VxWorks 6.7 BSP Manual

VxWorks-6.7 UT700 LEAP specific BSP manual

VXWORKS-6.7-UT700LEAP
Version 1.1.2.0
May 2014

Kungsgatan 12
411 19 Gothenburg
Sweden

tel +46 31 7758650
fax +46 31 421407
www.aeroflex.com/gaisler

VxWorks-6.7 BSP Manual

Copyright © 2014 Aeroflex Gaisler AB

Table of Contents

I. BSP 1
1. Introduction 2
1.1. Hardware 2
1.2. Source code 2
1.3. System clock 3
1.4. Fault tolerance 3
1.5. Clock gating unit 3
1.6. LCD Display 3
1.7. NVRAM 4
2. Examples 5
2.1. MIL-1553B 5
2.2. Spacewire 5
2.3. SPI 6
2.4. OC-CAN 6
3. Memory Configuration 7
3.1. Stack 7
4. Memory Management Unit 8
4.1. Initialization 8
4.2. Adding a virtual to physical map 8
5. Booting from persistent memory 9
5.1. LEON Boot loader 9
5.2. Booting VxWorks kernel from persistent memory 10
5.3. Bootrom 11
6. Interface 14
6.1. Function interface 14
6.2. VxWorks exception handling 14
7. Compilers 15
II. Drivers 16
8. VxBus infrastructure 17
8.1. Overview 17
8.2. Driver configuration 17
8.3. Source code 17
8.4. GRLIB AMBA Plug&Play bus 18
8.5. LEON2 AMBA bus 18
9. Timer driver interface 19
9.1. Overview 19
9.2. Source code 19
9.3. GPTIMER driver 19
9.4. LEON2 Timer driver 20
10. UART driver 21
11. Networking Support 22
11.1. Overview 22
11.2. Source code 22
11.3. Workbench Connection 22
11.4. Network drivers 22
12. GPIO Support 24
12.1. Overview 24
12.2. Source code 24
12.3. GPIO Library interface 24
12.4. GPIO Drivers 27
13. SPI Controller Driver 28
13.1. Overview 28
13.2. Source code 28
13.3. SPI Driver Interface 28
13.4. Example usage 31

14. OCCAN driver interface 33
14.1. CAN Hardware 33
14.2. Software Driver 33
14.3. Examples 33
14.4. Show routines 33
14.5. User interface 33
15. AHB Status register driver 41
15.1. Overview 41
15.2. Show routines 41
15.3. Source code 41
15.4. Operation 41
15.5. User interface 41
16. GR1553B Driver 43
16.1. Introduction 43
17. GR1553B Bus Controller Driver 45
17.1. Introduction 45
17.2. BC Device Handling 46
17.3. Descriptor List Handling 48
18. GR1553B Remote Terminal Driver 61
18.1. Introduction 61
18.2. User Interface 61
19. GR1553B Bus Monitor Driver 70
19.1. Introduction 70
19.2. User Interface 70
20. PCI Support 75
20.1. Overview 75
20.2. Source code 75
20.3. PCI Host drivers 75
20.4. PCI Board drivers 77
21. Support 79
22. Disclaimer 80

Part I. BSP

1. Introduction

This document provides documentation for the UT700 LEAP Board Support Package (BSP) for VxWorks 6.7. The BSP can be found in the `vxworks-6.7/target/config/gr_ut700leap` directory. For more information about the UT700 LEAP Board, see the user manual at http://www.aeroflex.com/ams/pagesproduct/datasheets/leon/LEAPUserManual.pdf.

1.1. Hardware

The supported hardware is summarized in the list below. Please see the drivers part of this document or the LEON VxWorks 6.7 Driver Manual for documentation about a specific core's driver.

- Interrupt controller
- UART console/terminal driver
- Timer unit
- General Purpose I/O (GRGPIO)
- 10/100 Ethernet networking (GRETH)
- SpaceWire (GRSPW)
- non-DMA CAN 2.0 (OCCAN)
- 1553: BC, RT and BM support (GR1553B)
- PCI support (GRPCI)
- Clock gating unit (GRCLKGATE)
- AHB Status Registers (AHBSTAT)

1.2. Source code

An overview of the source files in the BSP is given by the table below.

Table 1.1. BSP specific sources

Source	Description
sysLib.c	The BSP system library implementation. This is the heart of the BSP. The code handles reboot, system initialization in two stages, MMU setup and other needed functions.
sysALib.s	Low level setup, such as stack, trap table. Calls usrInit().
config.h	Default BSP configuration. This file contains the default project configuration when a project is created based upon the BSP. The settings can then be overridden from the Kernel Configuration GUI.
configNet.h	Networking configuration, number of network interfaces, and so on.
romInit.s	Boot loader implementation. Used to boot the bootrom (small VxWorks kernel with network boot capabilities) or standard kernel image. This contains the first instructions executed when starting from persistent memory. It initializes hardware registers such as the memory controller, CPU, FPU, wash memory etc. The settings are controlled by the configuration in bootcfg_def.h, config.h or project configuration (FOLDER_BOOTCFG in Hardware/BSP configuration variants/LEON ROM Boot setup).
bootcfg_def.h	Default boot loader configuration (used by romInit.s). It is included from config.h and the settings can be overridden from the project configurations, or a custom bootcfg_XYZ.h can be included instead of bootcfg_def.h.
bootloader.h	Used by the boot loader (romInit.s) to calculate values such as timer SCALER from system frequency.
Makefile	Makefile used when compiling BSP (often bootrom). One must configure which toolchain to use and the memory parameters to match config.h when creating a bootrom.

Source	Description
hwconf.c	VxBus Driver/Device configuration controlled from project settings.
*.cdf	BSP configuration files for WindRiver command line tools and Workbench kernel configuration GUI.
nvRam.c	Routines to manipulate non-volatile RAM which is accessed as normal RAM. For example MRAM.
sysSerial.c	Serial UART/Console routines, called from sysLib.c
lcd_st7565r	Basic driver for the LCD display included with the UT700 LEAP mezzanine.
sysNet	Provides support for the bootrom 'M' command to modify MAC addresses.

1.3. System clock

The BSP assumes a main clock frequency of 125MHz.

1.4. Fault tolerance

Error Detection And Correction (EDAC) is enabled by the boot loader, for example GRMON or the LEON boot loader. For GRMON this is done by starting with the "-edac" flag. See the GRMON manual for more information. For the LEON boot loader the BOOTCFG_FTMCTRL_MCFG3 define in `bootcfg_def.h` needs to be modified. It is also necessary to enable memory washing. For more information about the boot loader see Section 5.1. The AHBSTAT core can be used to detect and act on AMBA bus accesses triggering an error response. See the AHBSTAT driver, Chapter 15, for more information.

1.5. Clock gating unit

The clock gating unit can be used to enable and disable cores. By default all cores except the GR1553B are enabled. See Table 1.2. Cores can be enabled or disabled with GRMON using the "grcg enable *gate*" or "grcg disable *gate*" command. The clock gating unit can also be configured by the LEON boot loader. By default the boot loader enables all cores except the PCI and two of the Spacewire cores. See Table 1.2. These settings are more suitable to the UT700 LEAP board with mezzanine. The default boot loader clock gating can be easily changed, see Section 5.1.7, to enable the PCI and the two Spacewire cores if needed. The BSP contains the necessary PCI drivers.

Table 1.2. Default clock gating settings

Gate	Core	Description	UT700 Default	BSP Boot Loader Default
0	GRSPW0	Spacewire link 0	Enabled	Enabled
1	GRSPW1	Spacewire link 1	Enabled	Enabled
2	GRSPW2	Spacewire link 2	Enabled	Disabled
3	GRSPW3	Spacewire link 3	Enabled	Disabled
4	OCCAN	CAN core 1 & 2	Enabled	Enabled
5	GRETH	10/100 Ethernet MAC	Enabled	Enabled
6	GRPCI	32-bit PCI Bridge	Enabled	Disabled
7	GR1553B	MIL-STD-1553B	Disabled	Enabled

1.6. LCD Display

The UT700 LEAP mezzanine has a LCD display with a ST7565R controller. The BSP includes a simple driver to demonstrate its usage. It can be found under the *Device Drivers* folder under *UT700 LEAP BSP drivers*. By default the BSP will use it to display the Aeroflex logo when finished booting. To disable this behavior set *DRV_LCD_ST7565R_SHOW_LOGO* to *FALSE* in `config.h` or in the kernel configuration GUI.

1.7. NVRAM

By default the BSP uses the last 4kb of the MRAM to store boot settings, such as mac address and boot line. This can be disabled by setting *NVRAM_TYPE* to *NVRAM_TYPE_NONE* in `config.h`.

2. Examples

This chapter gives an overview of the example code provided with LEON VxWorks. For information on how to create a project, see the provided *Getting Started with LEON VxWorks* guide. The example code can be found in the installation directory. The examples are not BSP specific and might not be applicable to all hardware, or might require modification. They are mainly intended to demonstrate usage of different drivers.

2.1. MIL-1553B

This example demonstrates the usage of the GR1553B driver. It consist of two parts, one for creating a bus controller (BC) with a bus monitor (BM), and one for creating a remote terminal (RT) with a BM. Running the application requires two devices connected together by MIL-1553B. One of the devices needs to run the BC code and the other one the RT code.

2.1.1. Usage

Create two new image projects. Copy the files `pnp1553.h`, `config_bm.h` and `gr1553bcbm.c` to one of the projects and add the following lines to *usrAppInit* in `usrAppInit.c`.

```
int gr1553bcbm_test(void);
gr1553bcbm_test();
```

Include the DRV_GRLIB_GR1553BC and DRV_GRLIB_GR1553BM components in the kernel configuration. This project will take the role of the bus controller.

Copy the files `pnp1553.h`, `config_bm.h` and `gr1553rtbm.c` to the other project and add the following lines to *usrAppInit* in `usrAppInit.c`.

```
int gr1553rtbm_test(void);
gr1553rtbm_test();
```

Include the DRV_GRLIB_GR1553RT and DRV_GRLIB_GR1553BM components in the kernel configuration. This project will take the role of the remote terminal.

Executing the two images on two different devices connected by MIL-1553B will show the messages being exchanged between the devices.

2.2. Spacewire

This example takes commands from STDIN and generates SpaceWire packets with the path or logical address the user specififes. The application consists of three threads:

- TA01. Input task, the user commands from STDIN are interpreted into packet scheduling, time-code generation or status printing.
- TA02. Link monitor task. Prints out whenever a SpaceWire link switch from run-state to any other state or vice versa.
- TA03. SpaceWire DMA task. Handles reception and transmission of SpaceWire packets on all SpaceWire devices.

Tick-out IRQs are catched by the time-code ISR, and printed on STDOUT.

2.2.1. Usage

Create a new project and copy the files `grspw_pkt_lib.c`, `grspw_pkt_lib.h` and `grspw-test.c` to the project. Include the DRV_GRLIB_GRSPWPKT component in the kernel configuration. Add the following lines to *usrAppInit* in `usrAppInit.c`.

```
void spacewire_test(void);
spacewire_test();
```

The example allows commands to be entered to send packets along arbitrary path. Use the `h` command to display a list of available commands.

2.3. SPI

This example shows how to use the GRLIB SPI controller driver for single and periodic read/write transfers. The single transfer example demonstrates communication with a AD7814 temperature sensor and the periodic transfer example demonstrates communication with a AD7891 multi-channel AD converter.

2.3.1. Usage

Make sure that your board has the necessary hardware connected to the SPI controller. Create a new project and copy either the file `SPI-Single-AD7891.c` or the file `SPI-Periodic-AD7814.c` to the project. Include the DRV_GRLIB_SPICTRL component in the kernel configuration. Add the following lines to *usrAppInit* in `usrAppInit.c`.

```
/* For single transfers */
void printTemperature(void);
printTemperature();

/* For periodic transfers */
void printChannels(void);
printChannels();
```

The single transfer example will read and display a temperature reading with a small task delay. The periodic transfer example will repeatedly read the value of each channel from the AD converter and display the value with a small task delay.

2.4. OC-CAN

This example shows how to use the GRLIB OC-CAN driver to communicate with devices on a CAN bus. The example can be configured to act as a transmitter or receiver, or as both transmitter and receiver in loopback.

2.1. Usage

Connect two devices using CAN, or use one device in loopback mode. Create a new project and copy the files in the `occan` directory to the project. Include the DRV_GRLIB_OCCAN component in the kernel configuration. Add the following lines to *usrAppInit* in `usrAppInit.c`.

```
#include "occan_test.h"

/* For testing loopback mode */
occan_test_loopback();

/* To act as receiver */
occan_test_receiver();

/* To act as transmitter */
occan_test_transmitter();
```

The example will send and/or receive messages and display the message count.

3. Memory Configuration

The memory configuration is not auto detected in VxWorks. It is instead up to the user to configure the project with a valid memory configuration. The most important memory options are listed in the table below.

Table 3.1. BSP Memory parameters

Parameter	Description
LOCAL_MEM_LOCAL_ADRS	Start address of main memory. Default 0x40000000.
LOCAL_MEM_SIZE	Size of VxWorks available memory. Usually the memory size minus the size of the reserved memory. Default 0x02000000 for 32Mb RAM.
RAM_LOW_ADRS	Low RAM address. This is where the VxWorks image will be placed before jumping into sysALib.s. The trap table is installed on RAM_LOW_ADRS-0x3000. Default is the start address of the main memory + 0x3000, 0x40003000.
RAM_HIGH_ADRS	High RAM address. When the bootrom is used, the boot loader places the small VxWorks kernel (the bootrom) at high RAM. The RAM_LOW_ADRS..RAM_HIGH_ADRS is used by the bootrom kernel to store the VxWorks kernel fetched from the network before booting. Default set to half main memory + 0x3000, 0x41003000 on a system with 32Mb RAM.

3.1. Stack

The stack is located at top of the available main memory. Memory may be reserved over the stack making less memory available. The stack grows downwards to lower addresses on a SPARC CPU.

The Interrupt context is executed on a separate stack. That stack is located just above the SPARC trap table.

4. Memory Management Unit

The Memory Management Unit (MMU) offers memory protection between processes and kernel. Full protection can be achieved by not allowing processes to write to each others private memory or to the kernel area. If such a violation occurs the kernel halts the faulting process and the developer can find out what process and what instruction went wrong. The MMU is used by default, but can be disabled by removing the INCLUDE_MMU_BASIC component.

Physical addresses are mapped 1:1 into virtual address space. This is to make it easy to write device drivers that are independent of the MMU support.

4.1. Initialization

During most of the low level system startup the MMU is disabled. The BSP add areas that need to be mapped into virtual address space into the *sysPhysMemDesc* array. The most important area is of course the main memory. The areas mapped by the BSP using the *sysPhysMemDesc* array can be viewed by calling,

```
void sysMemMapShow (void);
```

The AMBA Plug&Play initialization routines map all APB Slave and AHB Slave I/O address spaces into virtual space on a 1:1 basis. This means that most of the on-chip AMBA drivers does not have to take into account if a MMU is present or not.

The PCI Host driver map all configured PCI BARs into virtual address space.

4.2. Adding a virtual to physical map

Physical addresses can be accessed directly using the sysIn/sysOut functions described in Section 6.1.1. Normally physical addresses does not have to be accessed directly. One can instead map the physical page (4k large) into virtual space by setting the appropriate permission bits. Pages can be mapped using platform independent VxWorks functions (vmLib.h), or by adding the map directly into the `sysLib.c`: sysPhysMemDesc[] array.

It is possible to debug the virtual to physical mapping using the GRMON command "walk 0xVIRT_ADDR" and "vmem 0xVIRT_ADDR".

5. Booting from persistent memory

A VxWorks image can be compiled to be started directly from RAM. This requires the image to be loaded into RAM by a hardware debugger, such as GRMON, or by a network boot loader, such as VxWorks bootrom. The image could alternatively be compiled to include a boot loader itself, allowing it to be started from a persistent memory such as FLASH/PROM/MRAM. The boot loader is responsible for setting up the memory controller configuration, and similar basic hardware initialization, and to copy the VxWorks kernel to RAM.

In this chapter the LEON boot loader and the bootrom is discussed. Note that when running the kernel from RAM using GRMON, it is not necessary to configure the boot loader since it is not used. GRMON itself does the basic initialization when connecting and typing *run*.

5.1. LEON Boot loader

The LEON boot loader is part of the BSP, in the file `romInit.s`. It initializes the memory controllers, CPU, FPU, washes memory etc. The boot loader's primary task is to load a VxWorks kernel. It performs basic initialization such as setting memory controller parameters and system console baud rate before booting VxWorks. The initialization parameters are configured using the VxWorks Kernel configuration utility from the Workbench, see Figure 5.1, or by modifying the default settings in the `bootcfg_def.h` file. Editing the defaults affect every newly created project based on the same BSP. Changes made using the Workbench only changes that particular project's boot loader settings.

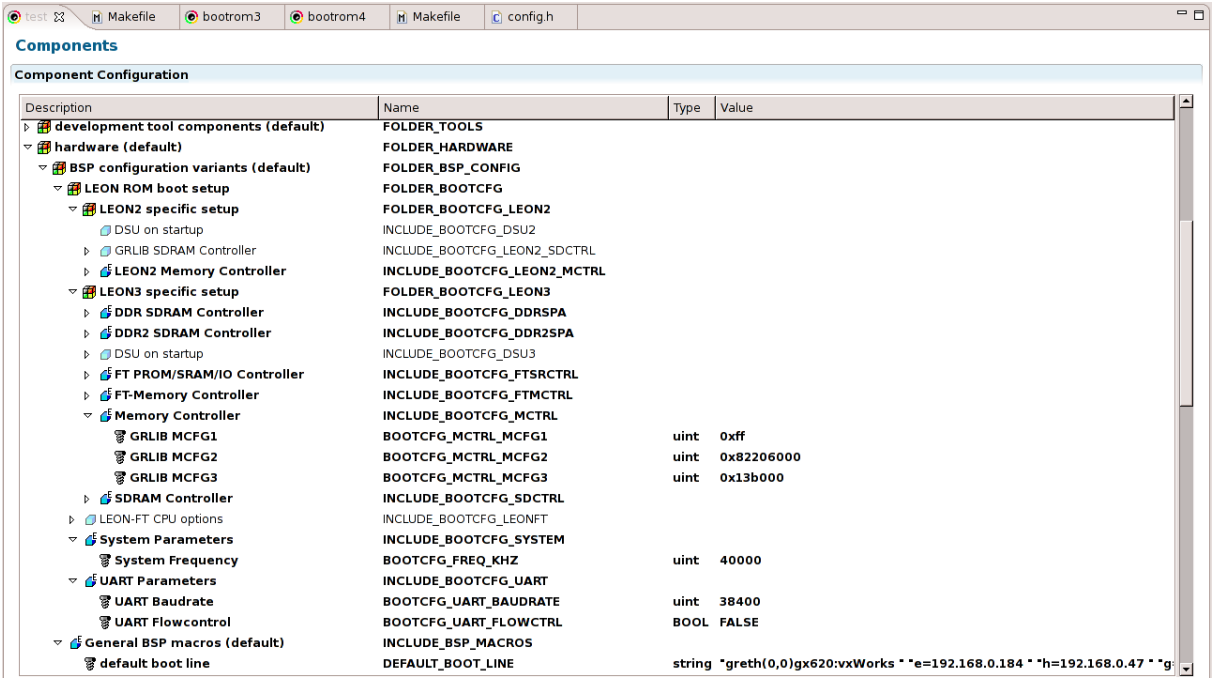

Figure 5.1. LEON Boot loader configuration

Some of the boot loader parameters are briefly described below. To find the correct initialization values, please consult the core documentation (GR-IP documentation).

5.1.1. BOOTCFG_FREQ_KHZ

Frequency of main AMBA bus in KHz. This setting affects the system timer interrupt frequency, baud rates calculated from the system frequency etc.

The default is 125000.

5.1.2. BOOTCFG_UART_BAUDRATE

The UART baud rate it is first initialized to before entering the VxWorks kernel.

The default is 38400 bits/second.

5.1.3. BOOTCFG_UART_FLOWCTRL

Enable or disable UART flow control on start up.

5.1.4. BOOTCFG_WASH_MEM

Enable or disable main memory clearing. All main memory available to VxWorks is washed by setting the contents to zero. Enabling washing can be necessary for fault tolerant systems. Any uninitialized data in the main memory might otherwise have incorrect check bits and will cause a trap when accessed the first time.

5.1.5. BOOTCFG_DSU3_ADDRESS

Debugging the application starting from persistent memory without GRMON can be hard sometimes. Using "grmon -ni" to connect into a target whose application has crashed may reveal much information. Even more can be extracted if the DSU trace buffers are enabled first thing during boot. The DSU is enabled early and is designed not to rely on the Plug&Play information, requiring the user to set BOOTCFG_DSU3_ADDRESS to the base address of the DSU hardware registers.

The default is to disable this feature. When enabled the default location of the DSU is 0x90000000.

5.1.6. Memory controller configuration options

The memory controller options are controlled by three register values, BOOTCFG_FTMCTRL_MCFG1, BOOTCFG_FTMCTRL_MCFG2 and BOOTCFG_FTMCTRL_MCFG3. The values are written to the memory controller's configuration registers. BOOTCFG_FTMCTRL_MCFG3 needs to be modified to enable EDAC. Default values are 0x1803c299, 0xF5B8600A and 0x083cd000.

5.1.7. Clock gating unit

The component INCLUDE_BOOTCFG_GRCG and the bitmask BOOTCFG_GRCG_ENABLED are used to tell the boot loader which cores it should enable or disable using the clock gating unit. Setting the least significant bit in the mask to one means that the core behind gate 0 should be enabled. A zero means that it should be disabled.

The hardware default is 0x7F, which enables all cores except the 1553. The BSP default is 0xB3, which better matches the capabilities of the LEAP board with mezzanine. See Table 1.2 in Section 1.5 for more details.

5.2. Booting VxWorks kernel from persistent memory

VxWorks kernels can be booted by using the LEON boot loader. The loader loads a VxWorks RAM image into the RAM_LOW_ADRS, and jumps into it. The configuration parameters of the boot loader must have been set up according to Section 5.1. The build rule can be changed from default to default_romCompress to create a boot loader including a compressed RAM image. The compressed image is extracted into RAM_LOW_ADRS.

5.2.1. Selecting build rule

The build rule can be selected using the Workbench. Choose build rule by pressing the right mouse button on the project icon and selecting 'Build Options/Set Active Build Spec...'. A dialog presents all possible targets for the project.

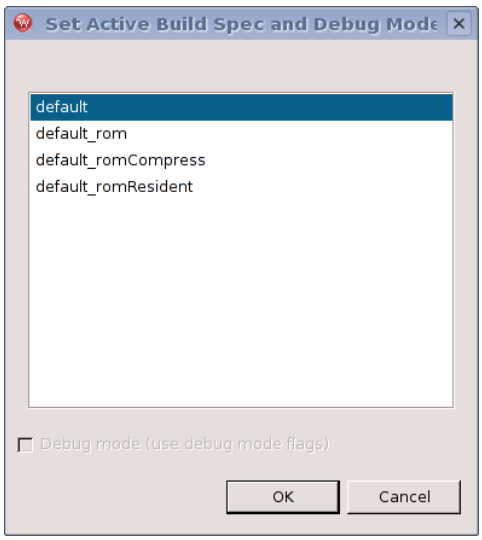

Figure 5.2. Workbench build rule dialog

The same targets apply for building projects with the command line tools. The build rule is passed as an environment variable ROM during project building. If ROM is left unset the default target will be built. Setting it to rom, romCompress or romResident selects one of the build rules described below.

- Default - image to run directly from RAM, no boot loader included. Typically used during debugging using grmon.
- Default_rom - image with boot loader. Programmed into persistent memory. Boot loader copys kernel to RAM and starts execution from there.
- Default_romCompress - *compressed* image with boot loader. Programmed into persistent memory. Boot loader decompress kernel into RAM and starts execution from there.
- Default_romResident - image intended to run from persistent memory. Programmed into persistent memory. Kernel is not copied to RAM. Used where RAM space is an issue.

5.3. Bootrom

This section describes the main steps in creating a VxWorks bootrom. The bootrom is typically used during the development process to download VxWorks kernels from a host on target reset and power on. The host can distribute VxWorks kernels over the network using FTP, TFTP and TSFS host file system. See the VxWorks documentation for more information.

A bootrom can be created using either the command line tools or the Workbench GUI. The make targets *vxworks-bsp-compile* and *vxworks-bsp-res-compile* have been prepared for compiling the bootrom using the command line tools.

The default bootrom is configured with networking and GRETH driver. The driver can be changed by editing `config.h`.

5.3.1. Configuring the boot loader

The bootrom is configured using the `config.h` file for component selection and `bootcfg_def.h` for custom boot loader parameters. From `config.h` and `Makefile` it is possible to do the memory configuration needed. See the defines ROM_SIZE, ROM_BASE_ADRS, RAM_LOW_ADRS, RAM_HIGH_ADRS, LOCAL_MEM_LOCAL_ADRS, LOCAL_MEM_SIZE in Chapter 3. The memory configuration must be the same for the boot loader and for VxWorks kernels loaded.

Depending on the boot method the component selection may vary. If networking is not needed or the LanChip driver must be used rather than the GRETH driver, it can be changed from within `config.h`.

The bootrom network and boot settings can be changed by editing DEFAULT_BOOT_LINE. The boot line argument definition can be found in the VxWorks documentation. Below is a typical boot line, booting the image VxWorks from host neptune (192.168.0.47) using the anonymous FTP service. The target itself has the IP-address 192.168.0.184 and name grxc3s1500.

```
greth0(0,0)neptune:vxWorks e=192.168.0.184 h=192.168.0.47 g=192.168.0.1 u=ftp pw=user f=0x04 tn=grxc3s1500
```

Hardware registers are initialized by the boot loader before the bootrom starts. The boot loader in `romInit.s` uses values from `bootloader.h` and `bootcfg_def.h` to set up the registers. One must create a custom `bootcfg_def.h` in order for the boot loader to successfully initialize the system. The boot loader parameters are described in Section 5.1.

5.3.2. Creating a bootrom project

A standard bootrom is created from the Workbench with the project creation guide, *File -> New -> VxWorks Boot Loader Project*. After giving the project a name, change the build target to Compressed or Resident and format to ELF.

The boot loader options are not available through the GUI as when creating VxWorks image projects. The configuration has to be done by hand, by editing the `config.h` file.

5.3.3. Building the bootrom

The memory configuration in `config.h` must match the memory configuration in `Makefile`. The toolchain used when building the the bootrom is controlled from the TOOL variable in `Makefile`. It can be changed to gnu/gnuv8/sfgnu/sfgnuv8 or diab/sfdiab.

Building the bootrom using the workbench is similar to building any other project, by pressing Build. The bootrom can be built using the different build targets bootrom, bootrom_uncmp, bootrom_res and bootrom_res_high. The default is bootrom. The bootrom target produces an image that will uncompress itself into RAM and run from RAM. The build target is selected when creating the project or by selecting "Project -> Build Options -> Set Active Build Spec". The bootrom ELF-file is created in the project directory, named bootrom.

When building using the command line tools, the make targets *vxworks-bsp-compile* and *vxworks-bsp-res-compile* has been prepared for compiling the bootrom using the command line tools. It is also possible to create a bootrom using the command line tools manually:

```
$ make execute
sh-3.00$ cd vxworks-6.7/target/config/gr_ut700leap/
sh-3.00$ make bootrom
```

5.3.4. Installing and running the bootrom

The bootrom can be installed into MRAM using GRMON.

```
grmon> load -wprot bootrom.prom
```

The wprot flag is necessary to temporarily disable the MRAM write protection while loading the image.

After a successful configuration the bootrom is booted after reset and power on. It can also be started from GRMON as follows,

```
grmon> run 0
```

The bootrom uses the serial terminal with the default settings as indicated by the table below. A terminal emulator can be started from within the workbench, "Window -> Show View -> Terminal". Other terminal emulators can also be used, for example minicom or hyper terminal. It is also possible to get the console output directly to GRMON's console if the flag -u is given when starting GRMON.

Table 5.1. Default terminal settings

Setting	Value
Baud rate	38400
Data bits	8
Stop bits	1
Parity	None
Flow control	None

5.3.5. Troubleshooting

When running the bootrom from GRMON works, but not when power cycling the board, it often is caused by bad memory controller settings. GRMON sets up them correctly, however the boot loader forgets to initializes them and everything seems to work.

Note that GRMON auto detects the memory controller configuration by using previous register contents for some bits. This means that if a faulty boot loader has been loaded, the boot loader may destroy the memory configuration and GRMON will not be able to auto detect memory parameters no more. To avoid this from happening the Gaisler boards have a *break* button which can be pressed during power cycling and reset that will prevent the CPU from start executing, it will break on reset.

One can compare the memory configuration with the configuration that GRMON auto detects, by issuing 'info sys' and 'info reg'. To avoid that GRMON initializes the registers on start up (overwriting the boot loader's memory configuration) the -ni flag can be used.

The BOOTCFG_DSUX_ADDRESS can be used to enable the DSU trace buffer on startup, enabling this during debugging may be helpful, when the boot loader crashes or hangs due to a faulty configuration, the last instructions may be viewed even though the application wasn't started by GRMON. When connecting with GRMON, the -ni flag can be used so that GRMON doesn't overwrite registers and memory, the instruction trace can be viewed by typing 'inst', the back trace can be helpful but it requires that the symbols are loaded with 'sym bootrom' for C function names to appear.

6. Interface

This section describes some of the LEON specific functions exported by the BSPs. The intention is not to document all functions. Many functions are documented in the BSP generic documents provided by WindRiver.

6.1. Function interface

6.1.1. sysOut* and sysIn*

The sysIn and sysOut functions can be used to load and store a 8-bit, 16-bit or 32-bit data to and from a physical address (the address is not translated). The loads and stores are guaranteed to occur in order. However, the LEON Level-1 cache is write-through, meaning all stores will always be directly written to memory (no cache effects as on some other platforms). sysIn* functions always skip the cache by using the LDA instruction to force a cache miss. This can be convenient sometimes when a core is doing DMA and the system does not have snooping, as a cache flush can be avoided.

For a non-MMU system all addresses are physical. On a MMU system the address will not be translated into a physical and the permission bits for that page will not have an effect. It will not cause a MMU trap when accessing an invalid page. For more information about mapping see Section 4.2.

6.2. VxWorks exception handling

The LEON port supports the standard VxWorks exception handling routines as described in the VxWorks Kernel Programmers Guide in the section Error Detection and Reporting.

SPARC exceptions and traps are described in the SPARC architecture manual. The ET bit in PSR enables/disables all kind of traps. Traps 0x80 and above are generated by software using "ta 0xTT" for system calls and enable/disable IRQ. When a trap is caused by an exception, the CPU will jump to the trap vector specific for the exception type. The trap table start address is set by the BSP at startup by writing to the trap base register (TBR). The lower bits of TBR indicate the last taken trap (the address of the last trap vector). The function *sparc_leon23_get_tbr()* can be used to get the TBR value.

The trap table is defined by the BSP in `sysALib.s`. The default action of unknown or unhandled traps is to enter excEnt().

The traps 0x02-0x04, 0x07-0x10 and 0x20-0x7f will cause a fatal exception. The default responses to a fatal exception are:

Table 6.1. Fatal exception handling

Type	**Deployed mode**	**Debug mode**	**Handler**
RTP	delete the RTP	stop the RTP	edrRtpFatalPolicyHandler
Kernel task	stop task	stop task	edrKernelFatalPolicyHandler
Init	reboot	reboot	edrInitFatalPolicyHandler
Interrupt	reboot	reboot	edrInterruptFatalPolicyHandler

It is possible to modify the the fatal exception handlers in `target/config/comps/src/edrStubs.c` to change the default behavior.

7. Compilers

The board support packages support both the DIAB and the GNU GCC compiler.

For information about a the compilers please see the LEON VxWorks compiler manual.

Part II. Drivers

This part of the BSP documentation provides details on the drivers applicable to the UT700 LEAP board. The content has been taken verbatim from the generic LEON VxWorks 6.7 Driver Manual. For this reason some of the sections might describe options that are not available on the UT700 LEAP board.

8. VxBus infrastructure

This section describes the SPARC/LEON specific VxBus bus controllers and device drivers in general.

The VxBus infrastructure is documented in the "VxWorks Device Driver Developer's Guide" document provided by WindRiver.

8.1. Overview

The VxBus infrastructure is used by the LEON BSPs and most of the LEON drivers. Legacy drivers are not affected by VxBus. All VxBus device drivers rely on a VxBus bus controller managing the bus that the hardware is situated on, in the LEON case the AMBA bus. Some of the drivers are PCI board drivers, they rely on the PCI bus controller driver. It is up to the bus driver to provide interrupt management.

Standard LEON2 uses hardcoded addresses and IRQ numbers of the cores on the AMBA bus, systems based on GRLIB uses the Plug&Play information to generate base addresses and IRQ numbers for the drivers. A LEON2 AMBA hardware driver can not be used with the LEON3 AMBA Plug&Play bus driver, or they way around. The LEON2 hardware addresses and IRQ numbers are configured in the `BSP/hwconf.c`, at the same location as the drivers are configured. Driver configuration parameters are described in the driver documentation.

8.2. Driver configuration

Most of the VxBus driver options are configured using driver resources in `BSP/hwconf.c`, they are usually controlled from the Kernel Configuration GUI or config.h. The resources targets a specific driver and one specific device that the driver operates. The driver is identified using a unique driver ID string and the bus type on which the driver is situated on. The device or core is identified using the unit number, the unit number is always the same which is normally assigned from the order in which they are found in Plug&Play.

Drivers normally request a resource during initialization, they can be used to override the driver defaults. The configuration can often be overridden later by using the driver interface.

8.3. Source code

Sources are located at `vxworks-6.7/target/src/hwif` and header files `vxworks-6.7/target/h/hwif`. Some drivers does not have an interface that is directly accessed by the user, they do not provide a header file. More specifically the LEON VxBus related sources are located as indicated by the table below, all paths are given relative to `vxworks-6.7/target`.

Table 8.1. LEON specific VxBus Bus controller sources

Location	Description
`src/hwif/vxbus/vxbPlb.c`	Processor Local Bus, hardcoded root bus used by LEON2 AMBA. The GRLIB AMBA bus is directly attached to the PLB bus, the PLB is nearly not used in this case.
`src/hwif/vxbus/vxbAmba.c`	GRLIB AMBA PnP bus controller driver's generic part. Multiple AMBA Plug&Play buses uses this generic part, on all GRLIB systems the on-chip AMBA Plug&Play bus uses it and on a RASTA system with a PCI board based on GRLIB the PCI board driver uses it as well.
`h/hwif/vxbus/vxbAmbaLib.h`	The GRLIB AMBA PnP bus controller driver's generic interface
`src/hwif/busCtrl/grlibAmba.c`	On-chip GRLIB AMBA PnP bus controller driver, this driver present the AMBA bus that the CPU is located at. Not used by LEON2, byte used by LEON2 in GRLIB. Remote AMBA buses
`config/BSP/hwconf.c`	LEON2 hardware addresses and IRQ. GRLIB and LEON2 driver configuration options per hardware device. This file is configured by the kernel configuration GUI, manually or indirect through config.h.

8.4. GRLIB AMBA Plug&Play bus

Systems based on GRLIB include an AMBA bus with Plug&Play information. The information is parsed and a device tree is created, each device in the tree is matched with a driver, if available. The GRLIB AMBA Plug&Play bus provides functions for pairing together an AHB Master, AHB Slave and an APB Slave into one AMBA "Core". This way the AMBA drivers are provided with all the information they need about the core, it is up the driver to decide what information is to be used.

In addition to the standard services such as interrupt management and hardware resources, the GRLIB bus controller provide the AMBA bus frequency and a address translation service for remote AMBA buses, for example an GRLIB AMBA bus accessed over PCI on a PCI board. The prototypes are shown below:

```
STATUS vxbAmbaFreqGet
  (
  struct vxbDev *pDev,      /* Device Information */
  unsigned int  *pFreqHz    /* Output Parameter */
  );

STATUS vxbAmbaTranslate
  (
  struct vxbDev *pDev,      /* Device Information */
  int fromLocal,            /* non-zero indicate that we translate from local (CPU bus) address
                             * to remote (Remote AMBA bus)
                             */
  unsigned int address,     /* Address to translate */
  unsigned int *result      /* Pointer to translated result. 0xffffffff is no map matching address */
  );
```

8.4.1. PCI Boards with AMBA Plug&Play bus

On a typical PCI board based on GRLIB, the PCI board driver implements an AMBA Plug&Play bus using the generic AMBA Plug&Play part to simplify implementation. The AMBA bus is often referred to as a "remote AMBA bus", it is not an on-chip bus. There are a couple of parameters that are different on a remote AMBA bus in comparison with the on-chip AMBA bus.

- Bus frequency.
- Interrupt management (always shared).
- Only parts of the bus is visible through "on-chip AMBA->PCI" window and then through "PCI->Remote AMBA" Window. This requires address translation.
- Other driver resources.

The interrupt management is different since it requires shared interrupts, due to the PCI boards may share interrupts.

8.5. LEON2 AMBA bus

The LEON2 AMBA bus implementation rely on the VxWorks generic Processor Local Bus (PLB) driver. It provides a hardcoded bus where all addresses and IRQ numbers must be specified in the `BSP/hwconf.c` configuration file.

8.5.1. GRLIB LEON2 systems

LEON2 systems that are base on GRLIB have an AMBA Plug&Play bus, however to maintain compatibility with the standard LEON2 peripherals the standard hardcoded LEON AMBA bus (PLB Bus) driver is used for the peripherals. On top of the PLB bus is the GRLIB AMBA Plug&Play bus driver providing support for the GRLIB cores. The LEON VxBus drivers for GRLIB will not recognize the difference and function as normal, thus both drivers can be used.

9. Timer driver interface

9.1. Overview

This section describes the timer drivers and interface specific for SPARC/LEON on VxWorks 6.7. There are two different timer drivers for the LEON BSPs, the GPTIMER and the LEON Timer driver. Both drivers follow the VxBus infrastructure, and is accessed using the WindRiver platform independent vxTimerLib. The vxTimerLib is in turn used through the sysClk*, sysTimestamp*, sysAuxClk and delay interfaces.

During initialization the vxTimerLib will try to allocate a number of timers to satisfy the requirement of the above mentioned interfaces. The first timer is used as system clock (sysClk), the second for user defined unused clock (auxClk), the third for as the time stamp clock and so on. If one two timers are available for example, the time stamp clock will be emulated using the system clock.

The timer system documentation is provided by WindRiver.

The GPTIMER and LEON2Timer PRESCALER register is initially initialized to eight by the AMBA bus routines found in `vxworks-6.7/target/src/drv/amba/ambaConfig.c`. The default PRESCALER value may be overridden by the drivers using driver resources. This means that to prescaler will provide the timers with a clock rate of SYSTEM_AMBA_FREQ/8 Hz.

9.2. Source code

Sources are located at `vxworks-6.7/target/src/hwif`. The LEON related timer driver sources are located as indicated by the table below, all paths are given relative to `vxworks-6.7/target`.

Table 9.1. LEON specific timer driver sources

Location	Description
`src/hwif/util/vxTimerLib.c`	The VxBus timer library, it uses the timer drivers to implement a platform independent interface.
`src/hwif/timer/grlibGpTimer.c`	GRLIB GPTIMER driver using VxBus. This driver supports multiple timer cores and multiple timers in each core.
`src/hwif/timer/leon2timer.c`	Standard LEON2 Timer core. Two timers are exported to the timer library.

9.3. GPTIMER driver

9.3.1. Hardware Support

The GPTIMER core can be configured in a number of ways, the GPTIMER driver in VxWorks 6.7 supports:

- Multiple timer cores
- Multiple timers within one core
- Non-Shared interrupts or shared interrupt between all timers

There is no special support for the optional watchdog functionality of the last timer of the GPTIMER core.

9.3.2. Using the driver

The GPTIMER driver is included by defining DRV_TIMER_GRLIB_GPTIMER either from the kernel configuration GUI or from config.h, the default is to include it. However if the system has another driver for creating timers, for example in a LEON2 system, the driver can safely be removed.

The GPTIMER driver has one configuration option, it can be set using the driver resources as described in Chapter 8. The driver can be configured to change the prescaler upon initialization, the prescaler value can

be controlled using the GPTIMER_SCALAR_RELOAD. The prescaler value can be changed in order to make the system clock or other timers/watchdogs more accurate.

9.4. LEON2 Timer driver

The LEON2 Timer driver supports the two timers available in hardware. The driver is included by defining DRV_TIMER_LEON2 either from the kernel configuration GUI or from config.h, the default is to include it. However if the system has another driver for creating timers, for example in a LEON3 system, the driver can safely be removed.

The driver has no configuration parameters.

10. UART driver

No specific documentation is needed for the UART driver, since the interface to the user is through the standard VxWorks console layer. Please see information about UART and termios in the VxWorks and POSIX documents.

The UART driver supports the LEON2 UART and the GRLIB APBUART core. The APBUART only support on-chip cores, not cores located at a PCI board such as the GR-RASTA-IO.

The system console is normally set to UART0, however it can be changed to UART1 by setting CONSOLE_TTY to 1. The baud rate of the VxWorks system console is controlled from CONSOLE_BAUD_RATE, note that when booting from FLASH/PROM the LEON boot loader will first initialize the baud rate according to the BOOTCFG_UART_BAUDRATE setting.

11. Networking Support

11.1. Overview

This section gives an introduction to the available Ethernet network drivers available in the SPARC/VxWorks 6.7 distribution. The network drivers are used from the VxWorks Networking stack and not by the user directly. The interface between the driver and the stack is documented in the WindRiver VxWorks documentation.

Networking applications are normally written using the standard BSD socket interface described in numerous books and on the Internet.

Two drivers are available, the SMC91C111/LanChip MAC driver for off-chip MAC solutions and a GRETH 10/100/100 driver for on-chip MAC solutions. Please read the documentation for the hardware in question.

11.2. Source code

Sources are located at as indicated in the table below, all paths are given relative to `vxworks-6.7/target/src/drv`.

Table 11.1. Ethernet MAC driver sources

Location	Description
amba/gaisler/greth/greth.c	GRETH 10/100/1000 Ethernet DMA MAC driver.
end/lan91c111End.c	The SMC91C111/LanChip 10/100 Ethernet MAC driver.

11.3. Workbench Connection

Both the Ethernet drivers supports the low level polling interface required by VxWorks to implement the communication channels to the WindRiver Workbench.

11.4. Network drivers

11.4.1. GRETH 10/100/1000 Ethernet MAC driver

The GRETH driver support both the GRETH 10/100 and the GRETH 10/100/GBit core. Since the GBit core has additional hardware features such as scatter gather, checksum off-loading and unaligned DMA access support the driver uses different functions to implement a as efficient strategy as possible. The driver is implemented using the VxBus infrastructure.

The GRETH driver is SMP-safe by using spin-locks. It has support for the VxWorks Ethernet multi-cast interface and uses the MII layer for PHY probing and initialization.

11.4.1.1. Adding and configuring

The GRETH Ethernet driver can be found under "Devices Drivers" in the VxWorks kernel configuration utility, or added from config.h by defining DRV_GRLIB_GRETH.

The driver has three configuration parameters, hardware MAC address, number of receive descriptors and the number of transmit descriptors. The MAC address must be unique on the local net. The number of descriptors has an impact on performance and memory usage. Each descriptor can hold one packet, the receive descriptor count is an important parameter to avoid packet loss. For example if the driver has been configured with 8 receive descriptors and 9 packets arrives so quickly that the driver hasn't had the time to process the incoming Ethernet packets, the hardware must drop the 9:th packet. On the other hand if a particularly problem maximally has 5 packets out at the same time, this may be the case of say two concurrent TCP/IP connections, it may be enough with 8 descriptors to avoid packet loss, thus saving about 1536 bytes per descriptor. Too few transmit descriptors may decrease performance, but does not result in packet loss.

In the GBit GRETH case, the GRETH writes directly into the buffer provided by the network stack, thus avoiding on extra copy (a zero-copy solution). Since no temporary buffers need to be used for the GBit core

the savings lowering the descriptor is just the descriptor it self, which is just 8 bytes. For the GBit core it is recommended to use the maximum number of descriptors.

11.4.1.2. Cache considerations

The GRETH core uses DMA to read and write Ethernet frames directly into main memory, for this to work the cache must be updated when Ethernet frames are written to memory by the GRETH. There are different methods that can be used to cope with unsynced caches,

- Cache flush on Ethernet Frame reception
- Cache snooping
- Mapping the Ethernet Frame non-cacheable using the MMU

The GRETH driver checks hardware if snooping is available, if it is, no action needs to be taken. However, if snooping is not available the cache is flushed before an Ethernet frame is handed over the VxWorks network stack.

Note that snooping is not enabled/disabled by the GRETH driver, the enable bit is just read.

Note that on a UT699 system snooping must be disabled for the GRETH driver to work. On the UT699 CPU the snooping enable bit can be set but has no effect, setting it will cause the GRETH driver to be fooled.

11.4.1.3. PHY interrupts

PHY interrupts can be used to detect Ethernet cable connections/disconnections, however older GRETH cores does not support it in hardware. The GRETH driver does not support PHY interrupt. A task in the platform independent VxWorks MII layer is polling all connected PHYs periodically.

11.4.2. SMC91C111/LANCHIP 10/100/1000 MAC driver

The LanChip driver can be found under "Devices Drivers" in the VxWorks kernel configuration utility, or added from config.h by defining INCLUDE_LAN91C111_END.

The MAC is accessed over the I/O bus, at address 0x20000300.

Note that the driver is not implemented using the VxBus infrastructure.

11.4.2.1. Configuring

The configuration parameters are listed in the table below. The PIO channel number is configurable via LA91C111, it is important for the interrupt generation.

Table 11.2. Lanchip driver configuration parameters

Parameter	Description
HWADDR	Configure Ethernet MAC address.
CFG_DUPLEX	Select duplex, 1 = Auto detect, 2 = Full Duplex, 3 = Half Duplex.
CFG_SPEED	Configure Ethernet speed, 1 = 100Mbps, 2 = 10Mbps, 3 = Auto detect.
PIO	PIO Channel number Lan91c111 chip interrupt line is connected, default is 4.

12. GPIO Support

12.1. Overview

This section gives an introduction to the available GPIO drivers and the simple GPIO Library that can be used to access the GPIO ports via the GPIO drivers.

12.2. Source code

Sources are located at as indicated in the table below, all paths are given relative to `vxworks-6.7/target`.

Table 12.1. GPIO Sources

Location	Description
h/hwif/grlib/gpiolib.h	GPIO Library user interface.
src/hwif/grlib/gpiolib.c	GPIO Library interface implementation sources.
src/hwif/grlib/grlibGrgpio.c	GRGPIO GPIO driver sources.

12.3. GPIO Library interface

This section describes the simple General Purpose Input/Output (GPIO) library interface for VxWorks 6.7. The GPIO Library implements a simple function interface that can be used to access individual GPIO ports. The GPIO Library provides means to control and connect an interrupt handler for a particular GPIO port. The library itself does not access the hardware directly but through a GPIO driver, for example the GRGPIO driver. A driver must implement a couple of function operations to satisfy the GPIO Library. The drivers can register GPIO ports during run time.

The two interfaces the GPIO Library implements can be found in the gpiolib header file (`hwif/grlib/gpiolib.h`), it contains definitions of all necessary data structures, bit masks, procedures and functions used when accessing the hardware and for the drivers implement GPIO ports.

The GPIO Library is included into the project by including the INCLUDE_DRV_GPIOLIB component from the Workbench kernel configuration GUI or by defining it in config.h.

This document describes the user interface rather than the driver interface.

12.3.1. Examples

There is an example avaible in the LEON VxWorks distribution at `usr/gr_rasta_adcdac_nommu1/gpio-demo.c`. The example has been designed to be operated on an AMBA-over-PCI bus. The example needs to be modified if the GRGPIO used in located on-chip.

12.3.2. Driver interface

The driver interface is not described in this document.

12.3.3. User interface

The GPIO Library provides the user with a function interface per GPIO port. The interface is declared in gpiolib.h. GPIO ports are registered by GPIO drivers during run time, depending on the registration order the GPIO port are assigned a port number. The port number can be used to identify a GPIO port. A GPIO port can also be referenced by a name, the name is assigned by the GPIO driver and is therefore driver dependent and not documented here.

The interface can be directly accessed using the VxWorks C Shell.

GPIO ports which does not support a particular feature, for example interrupt generation, return error codes when tried to be accessed.

12.3.3.1. Accessing a GPIO port

The interface for one particular GPIO port is initialized by calling *gpioLibOpen* with a port number or *gpioLibOpenByName* with the device name identifying one port. The functions returns a pointer used when calling other functions identifying the opened GPIO port. If the device name can not be resolved to a GPIO port the open function return NULL. The prototypes of the initialization routines are shown below:

```
void *gpioLibOpen(int port)
void *gpioLibOpenByName(char *devName)
```

Note that this function must be called first before accessing other functions in the interface.

Note that the port naming is dependent of the GPIO driver used to access the underlying hardware.

12.3.3.2. Interrupt handler registration

Interrupt handlers can be installed to handle events as a result to GPIO pin states or state changes. Depending on the functions supported by the GPIO driver, four interrupt modes are available, edge triggered on falling or rising edge and level triggered on low or high level. It is possible to register a handler per GPIO port by calling *gpioLibIrqRegister* setting the arguments correctly as described in the table below is the prototype for the IRQ handler (ISR) install function.

```
int gpioLibIrqRegister(void *handle, VOIDFUNCPTR func, void *arg)
```

The function takes three arguments described in the table below.

Table 12.2. gpioLibIrqRegister argument description

Name	Description
handle	Handle used internally by the function interface, it is returned by the open function.
func	Pointer to interrupt service routine which will be called every time an interrupt is generated by the GPIO hardware.
arg	Argument passed to the func ISR function when called as the second argument.

To enable interrupt, the hardware needs to be initialized correctly, see functions described in the function prototype section. Also the interrupts needs to be unmasked.

12.3.3.3. Data structures

The data structure used to access the hardware directly is described below. The data structure *gpiolib_config* is defined in *gpiolib.h*.

```
struct gpiolib_config {
    char mask;
    char irq_level;
    char irq_polarity;
}
```

Table 12.3. gpiolib_config member description.

Member	Description	
mask	Mask controlling GPIO port interrupt generation	
	0	Mask interrupt
	1	Unmask interrupt
irq_level	Level or Edge triggered interrupt	

Member	Description	
	0	Edge triggered interrupt
	1	Level triggered interrupt
irq_polarity	Polarity of edge or level	
	0	Low level or Falling edge
	1	High level or Rising edge

12.3.3.4. Function prototype description

A short summary to the functions are presented in the prototype lists below.

Table 12.4. GPIO per port function prototypes

Prototype Name
void gpioLibClose(void *handle)
int gpioLibSetConfig(void *handle, struct gpiolib_config *cfg)
int gpioLibSet(void *handle, int dir, int val)
int gpioLibGet(void *handle, int *inval)
int gpioLibIrqClear(void *handle)
int gpioLibIrqEnable(void *handle)
int gpioLibIrqDisable(void *handle)
int gpioLibIrqForce(void *handle)
int gpioLibIrqRegister(void *handle, VOIDFUNCPTR func, void *arg)
void gpioLibShowInfo(int port, void *handle)
void gpioLibShow(void)

All functions takes a handle to an opened GPIO port by the argument *handle*. The handle is returned by the *gpioLibOpen* or *gpioLibOpenByName* function.

If a GPIO port does not support a particular operation, a negative value is returned. On success a zero is returned.

12.3.3.4.1. gpioLibSetConfig

Configures one GPIO port according to the the gpiolib_config data structure.

The gpiolib_config structure is described in Table 12.3.

12.3.3.4.2. gpioLibSet

Set one GPIO port in output or input mode and set the GPIO Pin value. The direction of the GPIO port is controlled by the *dir* argument, 1 indicates output and 0 indicates input. The third argument is not used when *dir* is set to input. The value driven by the GPIO port is controlled by the *val* argument, it is driven low by setting *val* to 0 or high by setting *val* to 1.

12.3.3.4.3. gpioLibGet

Get the input value of a GPIO port. The value is stored into the address indicated by the argument *inval*.

12.3.3.4.4. gpioLibIrqClear

Acknowledge any interrupt at the interrupt controller that the GPIO port is attached to. This may be needed in level sensitive interrupt mode.

12.3.3.4.5. gpioLibIrqEnable

Unmask GPIO port interrupt on the interrupt controller the GPIO port is attached to. This enables GPIO interrupts to pass though to the interrupt controller.

12.3.3.4.6. gpioLibIrqDisable

Mask GPIO port interrupt on the interrupt controller the GPIO port is attached to. This disable interrupt generation at the interrupt controller.

12.3.3.4.7. gpioLibIrqForce

Force an interrupt by writing to the interrupt controller that the GPIO port is attached to.

12.3.3.4.8. gpioLibIrqRegister

Attaches an interrupt service routine to a GPIO port. Described separately in section Section 12.3.3.2.

12.3.3.4.9. gpioLibShowInfo

Show the current status one or all GPIO ports in the system. Integer 'port' is port number, if port = -1 all ports are selected. If port != -1, handle is used to get port. If port != -1, handle == NULL, then port is used as port number.

12.3.3.4.10. gpioLibShow

Shows all GPIO ports in the system.

12.4. GPIO Drivers

The GPIO drivers listed here can be used together with the GPIO Library. The interface to the user is thus the same independent of GPIO driver. However, depending on what the hardware supports features may be disabled, for example all GPIO pins may not support interrupt generation.

12.4.1. GRGPIO driver

This section describes the GRGPIO driver for VxWorks 6.7, it is used by accessing the GPIO Library documented in another section.

The GRGPIO driver is included into the project by including the INCLUDE_DRV_GRGPIO component from the Workbench kernel configuration GUI or by defining it in config.h.

A GRGPIO port is accessed using an unique port number or by name. The name of a GRGPIO port is `/grgpio/CORE_NUM/PORT_NUM`, and on a PCI board `PCI_BOARD/BOARD_NUM/grgpio/CORE_NUM/PORT_NUM`, where CORE_NUM is the number of the GRGPIO on the AMBA bus, PORT_NUM is the GPIO port/pin of that GRGPIO core, PCI_BOARD the name of the PCI system (/gr_rasta_io, /gr_701), and BOARD_NUM is the number of the PCI board of that type. Accessing pin 13 on GRGPIO core two on the first GR-RASTA-IO PCI board would be `/gr_rasta_io/0/grgpio/1/13`.

The GRGPIO core is configurable, some cores implement the BYPASS register which enables the user to select between GPIO or some other core's functionality for a specific pin. The BYPASS register can be set using the GRGPIOX_BYPASS parameter. The GRGPIO core can also be configured with different number of ports, the driver can auto detect this by writing DIR and OUTPUT registers, however that may cause previously configured GPIO port to be destroyed, in that case auto detection may be avoided by the GRGPIOX_NUM_PORTS parameter.

13. SPI Controller Driver

13.1. Overview

This section gives an introduction to the SPI driver. The driver supports both standard and periodic mode. Periodic mode is not available for all SPI controller cores.

13.2. Source code

Sources are located as indicated in the table below. All paths are given relative to `vxworks-6.7/target`.

Table 13.1. SPI Sources

Location	Description
h/hwif/grlib/grlibSpictrl.h	SPI driver user interface.
src/hwif/grlib/grlibSpictrl.c	SPI driver source.

13.3. SPI Driver Interface

13.3.1. Function prototype description

A short summary to the functions are presented in the prototype lists below.

Table 13.2. SPI driver function prototypes

Prototype Name	Description
grlibspictrl_handle grlibspictrl_get_handle(int minor)	Returns a handle to SPI controller SPI{minor}
unsigned int spictrl_status(grlibspictrl_handle handle)	Return the status of the SPI controller (the event register)
int spictrl_configure(grlibspictrl_handle handle, unsigned int bitrate, unsigned char bits_per_word, char lsb_first, char clock_inv, char clock_phs, unsigned int idle_word, unsigned int* effective_bitrate)	Configures the SPI controller
int spictrl_read_write_bytes(grlibspictrl_handle handle, void *rxbuf, void *txbuf, int nbytes)	Performs a SPI transfer
int spictrl_read_bytes(grlibspictrl_handle handle, unsigned char *bytes, int nbytes)	Performs a read-only SPI transfer
int spictrl_write_bytes(grlibspictrl_handle handle, unsigned char *bytes, int nbytes)	Performs a write-only SPI transfer
int spictrl_send_addr(grlibspictrl_handle handle, uint32_t addr, int rw)	Enables a SPI slave
int spictrl_send_stop(grlibspictrl_handle handle)	Disables all SPI slaves
int spictrl_configure_periodic(grlibspictrl_handle handle, int clock_gap, unsigned int flags, int periodic_mode, unsigned int period, unsigned int period_flags, unsigned int period_slvsel)	Configures periodic transfers
int spictrl_start_periodic(grlibspictrl_handle handle)	Starts periodic transfers
void spictrl_stop_periodic(grlibspictrl_handle handle)	Stops periodic transfers
int spictrl_write_periodic(grlibspictrl_handle handle, void *txbuf, unsigned int nbytes)	Set data to write each periodic transfer

Prototype Name	Description
int spictrl_read_periodic(grlibspictrl_handle handle, void *rxbuf, unsigned int nbytes)	Read data from last periodic transfer
int spictrl_set_buffer_mask_periodic(grlibspictrl_handle handle, unsigned int mask[4])	Set buffer mask for periodic transfers
int spictrl_buffer_length_periodic(grlibspictrl_handle handle, int words)	Set buffer length for periodic transfers

All functions requires a *handle* to a specific SPI core. The handle is created by the *grlibspictrl_get_handle* function.

13.3.1.1. grlibspictrl_get_handle

Returns a handle to the SPI core with index *minor*. If no such core can be found the function returns 0.

13.3.1.2. spictrl_status

Return the status of the SPI controller (the event register).

13.3.1.3. spictrl_configure

Configures the SPI controller for commmunication. Returns 0 if successful or -1 if the bitrate was too low.

Table 13.3.

Name	Type	Parameter description
bitrate	unsigned int	The requested bitrate.
bits_per_word	unsigned int	This value determines the length in bits of a transfer on the SPI bus. Legal values are 4-16 or 32. The value must not be greater than the maximum allowed word length specified by the MAXWLEN field in the capability register.
lsb_first	char	When set to 0 the data is transmitted LSB first. When it is non-zero the data is transmitted MSB first.
clock_inv	char	Determines the polarity (idle state) of the SCK clock. Can be set to 0 or 1.
clock_phs	char	When set to 0 the data will be read on the first transition of SCK. When given a non-zero value the data will be read on the second transition of SCK.
idle_word	unsigned int	For read only operations the write part of the transaction will consist of duplicates of this word.
effective_bitrate	unsigned int*	Pointer to a variable that will contain the effective bitrate. Will be as close to the requested bitrate as is possible by the hardware. Can be set to 0 if the value is not needed.

13.3.1.4. spictrl_read_write_bytes

Performs a SPI transaction. *nbytes* specifies the length of the transaction in bytes. If *txbuf* is set to 0 the write part of the transaction will consist of the *idle_word* given when configuring the controller.

Returns the number of bytes transfered or -1 if the core was configured for periodic mode.

13.3.1.5. spictrl_read_bytes

Performs a read only SPI transaction. *nbytes* specifies the length of the transaction in bytes. The write part of the transaction will consist of the *idle_word* given when configuring the controller.

Returns the number of bytes transfered or -1 if the core was configured for periodic mode.

13.3.1.6. spictrl_write_bytes

Performs a write only SPI transaction. *nbytes* specifies the length of the transaction in bytes.

Returns the number of bytes transfered or -1 if the core was configured for periodic mode.

13.3.1.7. spictrl_send_addr

Activate slave select signal with index *addr*. Can be overridden by a custom slave select function. See !!!!!.

Returns 0 if successful or -1 if the address is higher than the number of available slave select signals. If the function is overridden the return value is decided by the custom slave select function.

13.3.1.8. spictrl_send_stop

Inactivate all slave select signals.

Returns 0. If the slave select function is overridden the return value is decided by the custom slave select function.

13.3.1.9. spictrl_configure_periodic

Configures the SPI controller for periodic transactions. This functionality is not available on all SPI cores.

Returns 0 if successful, -1 if periodic mode has already been started, or -2 if periodic mode is not supported by the hardware.

Table 13.4.

Name	Type	Parameter description
clock_gap	unsigned int	The core will insert *clock_gap* SCK clock cycles between each consecutive word. Value should be lower than 32.
flags	unsigned int	Can be set to 0 or SPICTRL_FLAGS_TAC. The latter enables automatic slave select during clock gap. This causes the core to perform the swap of the slave select registers at the start and end of each clock gap.
periodic_mode	unsigned int	Set to 1 to enable automatic periodic mode.
period	unsigned int	Number of clocks between automated transfers. Maximum value is implementation dependent.
period_flags	unsigned int	Possible flags are *SPICTRL_PERIOD_FLAGS_EACT* and *SPICTRL_PERIOD_FLAGS_ASEL*. The first will enable external activation of automated transfers while the latter enables automatic slave select.
period_slvsel	unsigned int	Slave select when transfer is not active. Default is 0xffffffff.

13.3.1.10. spictrl_start_periodic

Start performing periodic SPI transactions.

Returns 0.

13.3.1.11. spictrl_stop_periodic

Stop performing periodic SPI transactions.

13.3.1.12. spictrl_write_periodic

Set data to be written in each periodic SPI transaction.

Returns 0 if successful.

13.3.1.13. spictrl_read_periodic

Get data read in the last periodic SPI transaction.

Returns 0 if successful.

13.3.1.14. spictrl_set_buffer_mask_periodic

Sets a bit mask that determines which words in the AM Transmit / Receive queues to read from / write to. Bit 0 of the first mask word corresponds to the first position in the queues, bit 1 of the first mask word to the second position, bit 0 of the second mask word corresponds to the 33:d position, etc. The total number of bits implemented equals FDEPTH (bit 15:8) in the SPI controller capability register.

Use either this function or *spictrl_buffer_length_periodic* to specify the length of the periodic SPI transaction.

Returns 0 if successful or -1 if the number of words used is larger than FDEPTH.

13.3.1.15. spictrl_buffer_length_periodic

Sets the length of the periodic SPI transaction in words.

Use either this function or *spictrl_set_buffer_mask_periodic* to specify the length of the periodic SPI transaction.

Returns 0 if successful or -1 if the number of words used is larger than FDEPTH.

13.4. Example usage

To use the SPI controller driver it needs to be included in the kernel configuration. The name of the driver is *DRV_GRLIB_SPICTRL*. To access the actual driver functions one needs to include the SPI controller driver header file:

```
#include <hwif/grlib/grlibSpictrl.h>
```

A handle can then be acquired to a SPI core. The following function call returns a handle for SPI controller SPI0:

```
spihandle = grlibspictrl_get_handle(0 /*SPI0*/);
```

In the remainder of the example we will assume that SPI0 is connected to an AD7814 temperature sensor. To configure the SPI controller we call *spictrl_configure* with the following parameters:

```
spictrl_configure(spihandle, 200000, 16, 0, 1, 1, 0, 0);
```

The first argument is the handle to the SPI0 core. The second is the requested bitrate. For the AD7814 the bits per word is 16, the MSB is sent first, the polarity is 1 and the data is read on the second transition. The idle word is 0 and we are not interested in the effective bitrate. Once the core is configured we use *spictrl_send_addr* to enable the temperature sensor using slave select.

```
spictrl_send_addr(spihandle, 0, 0);
```

We can now perform a SPI transfer where write the idle word and read the temperature.

```
spictrl_read_bytes(spihandle, (unsigned char*)&temperature, 2);
```

Once we have finished the transfer we disable the temperature sensor using slave select.

```
        spictrl_send_stop(spihandle);
```

14. OCCAN driver interface

This section describes the OCCAN CAN 2.0 core driver for VxWorks 6.7.

14.1. CAN Hardware

The OC_CAN core can operate in different modes providing the same register interfaces as other well known CAN cores. The OC_CAN driver supports PeliCAN mode only.

The driver supports the two different register MAPs, standard 8-bit or non-standard 32-bit. The driver must be compiled with either a WORD or a BYTE aligned register MAP. The both cannot be mixed in the same system. WORD alignment can be selected using the OCCAN_WORD_REGS parameter and by editing `src/drv/grdrv/occan/grdrv_occan.c`.

This CAN hardware does not support DMA. One interrupt is taken per transmitted and received CAN message.

14.2. Software Driver

The driver provides means for processes and threads to send and receive messages. Errors can be detected by polling the status flags of the driver. Bus off errors cancels the ongoing transfers to let the caller handle the error.

The driver supports filtering received messages id fields by means of acceptance filters, runtime timing register calculation given a baud rate. However not all baud rates may be available for a given system frequency. The system frequency is hard coded and must be set in the driver.

14.3. Examples

There are currently no example available for OCCAN.

14.4. Show routines

During debugging it is sometimes to see the status of the driver and hardware. The OCCAN driver provides two functions listed below that can be used directly from the VxWorks C shell or in an application. The OCCAN_SHOW_ROUTINES parameter must be defined during compile time to include the show routines.

```
void occanShow(int short_info)
void occanShowRegs(pelican_regs *regs)
```

Driver information of all OCCAN devices can be printed with *occanShow*, if the argument is set to a non-zero value the function will also print RX/TX statistics, OCCAN registers, and RX/TX fifo status.

The OCCAN registers can be listed by calling *occanShowRegs* with the base register address of the OCCAN core. This call is not dependent of the driver.

14.5. User interface

The VxWorks OCCAN driver supports the standard accesses to file descriptors such as *read*, *write* and *ioctl*. User applications include the grcan driver's header file (grlibOccan.h) which contains definitions of all necessary data structures and bit masks used when accessing the driver.

The GRCAN driver is implemented using the VxBus infrastructure.

14.5.1. Driver registration

The registration of the driver is crucial for threads and processes to be able to access the driver using standard means, such as open. The VxWorks I/O driver registration is performed automatically by the driver when CAN hardware is found for the first time. The driver is called from the VxBus infrastructure to handle detected CAN hardware. In order for the VxBus to match the CAN driver with the CAN hardware one must register the driver. This process is automatically done when including the driver into the project.

14.5.2. Driver resource configuration

The driver can be configured using driver resources as described in the VxBus chapter. Below is a description of configurable driver parameters. The driver parameters are unique per CAN device. The parameters are all optional, the parameters only overrides the default values. The resources can be configured from the Workbench kernel configuration GUI.

Table 14.1. OCCAN driver parameter description

Name	Type	Parameter description
txFifoLen	INT	Length of TX software FIFO. The FIFO is used when then hardware FIFO is full to store up work. The FIFO is emptied from the interrupt handler when a CAN message has been successfully transmitted.
rxFifoLen	INT	Length of RX software FIFO. Every received CAN message is put into the receive FIFO, from the interrupt handler. The FIFO is emptied by the user in the read() call.

14.5.3. Opening the device

Opening the device enables the user to access the hardware of a certain OC_CAN device. The driver is used for all OC_CAN devices available. The devices is separated by assigning each device an unique name and a number called minor. The name is passed during the opening of the driver.

Table 14.2. Device number to device name conversion.

Device number	Filesystem name	Location
0	/occan/0	On-Chip Bus
1	/occan/1	On-Chip Bus
2	/occan/2	On-Chip Bus
Depends on system configuration	/gr701/0/grcan/0	GR-RASTA-IO

An example VxWorks open call is shown below.

```
fd = open("/dev/occan0", O_RDWR)
```

A file descriptor is returned on success and -1 otherwise. In the latter case errno is set as indicated in Table 14.3.

Table 14.3. Open errno values.

ERRNO	Description
ENODEV	Illegal device name or not available
EBUSY	Device already opened
ENOMEM	Driver failed to allocate necessary memory

14.5.4. Closing the device

The device is closed using the close call. An example is shown below.

```
res = close(fd)
```

Close always returns 0 (success) for the occan driver.

14.5.5. I/O Control interface

Changing the behavior of the driver for a device is done via the standard system call *ioctl*. Two arguments must be provided to ioctl, the first being an integer which selects ioctl function and secondly a pointer to data that may be interpreted uniquely for each function. A typical ioctl call definition:

```
int ioctl(int fd, int cmd, void *arg);
```

The return value is 0 on success and -1 on failure and the global errno variable is set accordingly.

All supported commands and their data structures are defined in the OCCAN driver's header file `grlibOccan.h`. In functions where only one argument is needed the pointer (void *arg) may be converted to an integer and interpreted directly, thus simplifying the code.

14.5.6. Data structures

The occan_afilter structure is used when changing acceptance filter of the CAN receiver.

```
struct occan_afilter {
    unsigned int code[4];
    unsigned int mask[4];
    int single_mode;
};
```

Table 14.4. occan_afilter member description.

Member	Description
code	Specifies the pattern to match, only the unmasked bits are used in the filter.
mask	Selects what bits in *code* will be used or not A set bit is interpreted as don't care.
single_mode	Set to none-zero for a single filter - single filter mode, zero selects dual filter mode.

The CANMsg struct is used when reading and writing messages. The structure describes the driver's view of a CAN message. The structure is used for writing and reading. The *sshot* fields lacks meaning during reading and should be ignored. See the transmission and reception section for more information.

```
typedef struct {
    char extended;
    char rtr;
    char sshot;
    unsigned char len;
    unsigned char data[8];
    unsigned int id;
} CANMsg;
```

Table 14.5. CANMsg member description.

Member	Description
extended	Indicates whether message has 29 or 11 bits ID tag. Extended or Standard frame.
rtr	Remote Transmission Request bit.
sshot	Single Shot. Setting this bit will make the hardware skip resending the message on transmission error.
len	Length of data.
data	Message data, data[0] is the most significant byte - the first byte.
Id	The ID field of the message. An extended frame has 29 bits whereas a standard frame has only 11-bits. The most significant bits are not used.

The occan_stats data structure contains various statistics gathered by the OCCAN hardware.

```
typedef struct {
    /* tx/rx stats */
    unsigned int rx_msgs;
    unsigned int tx_msgs;

    /* Error Interrupt counters */
    unsigned int err_warn;
    unsigned int err_dovr;
```

```
        unsigned int err_errp;
        unsigned int err_arb;
        unsigned int err_bus;

        /* ALC 4-0 */
        unsigned int err_arb_bitnum[32];

        /* ECC 7-6 */
        unsigned int err_bus_bit; /* Bit error */
        unsigned int err_bus_form; /* Form Error */
        unsigned int err_bus_stuff; /* Stuff Error */
        unsigned int err_bus_other; /* Other Error */

         /* ECC 5 */
        unsigned int err_bus_rx;
        unsigned int err_bus_tx;

        /* ECC 4:0 */
        unsigned int err_bus_segs[32];

        /* total number of interrupts */
        unsigned int ints;

        /* software monitoring hw errors */
        unsigned int tx_buf_error;

    } occan_stats;
```

Table 14.6. occan_stats member description.

Member	Description
rx_msgs	Number of CAN messages received.
tx_msgs	Number of CAN messages transmitted.
err_warn	Number of error warning interrupts.
err_dovr	Number of data overrun interrupts.
err_errp	Number of error passive interrupts.
err_arb	Number of times arbitration has been lost.
err_bus	Number of bus errors interrupts.
err_arb_bitnum	Array of counters, err_arb_bitnum[index] is incremented when arbitration is lost at bit index.
err_bus_bit	Number of bus errors that was caused by a bit error.
err_bus_form	Number of bus errors that was caused by a form error.
err_bus_stuff	Number of bus errors that was caused by a stuff error.
err_bus_other	Number of bus errors that was not caused by a bit, form or stuff error.
err_bus_tx	Number of bus errors detected that was due to transmission.
err_bus_rx	Number of bus errors detected that was due to reception.
err_bus_segs	Array of 32 counters that can be used to see where the frame transmission often fails. See hardware documentation and header file for details on how to interpret the counters.
ints	Number of times the interrupt handler has been invoked.

14.5.7. Configuration

The CAN core and driver are configured using *ioctl* calls. The Table 14.8 below lists all supported ioctl calls. OCCAN_IOC_ must be concatenated with the call number from the table to get the actual constant used in the code. Return values for all calls are 0 for success and -1 on failure. Errno is set after a failure as indicated in Table 14.7.

An example is shown below where the receive and transmit buffers are set to 32 respective 8 by using an ioctl call:

```
result = ioctl(fd, OCCAN_IOC_SET_BUFLEN, (8 << 16) | 32);
```

Table 14.7. Ioctl errno values.

ERRNO	Description
EINVAL	Null pointer or an out of range value was given as the argument.
EBUSY	The CAN hardware is not in the correct state. Many ioctl calls need the CAN device to be in reset mode. One can switch state by calling START or STOP.
ENOMEM	Not enough memory to complete operation. This may cause other ioctl commands to fail.

Table 14.8. ioctl calls supported by the CAN driver.

Call Number	Call Mode	Description
START	Reset	Exit reset mode, enter operating mode. Enables read and write. This command must be executed before any attempt to transmit CAN messages on the bus.
STOP	Running	Exit operating mode, enter reset mode. Most of the settings can only be set when in reset mode.
GET_STATS	Don't care	Get current statistics collected by driver.
GET_STATUS	Don't care	Get the status register. Bus off among others can be read out.
SET_SPEED	Reset	Set baud rate, the timing values are calculated using the core frequency and the location of the sampling point.
SET_BTRS	Reset	Set timing registers manually.
SET_BLK_MODE	Don't care	Set read and write blocking/non-blocking mode
SET_BUFLEN	Reset	Set receive and transmit software FIFO buffer length.
SET_FILTER	Reset	Set acceptance filter. Let the second argument to the ioctl command point to a *occan_afilter* data structure.

14.5.7.1. START

This ioctl command places the CAN core in operating mode. Settings previously set by other ioctl commands are written to hardware just before leaving reset mode. It is necessary to enter running mode to be able to read or write messages on the CAN bus.

The command will fail if receive or transmit buffers are not correctly allocated or if the CAN core already is in operating mode.

14.5.7.2. STOP

This call makes the CAN core leave operating mode and enter reset mode. After calling STOP further calls to read and write will result in errors.

It is necessary to enter reset mode to change operating parameters of the CAN core such as the baud rate and for the driver to safely change configuration such as FIFO buffer lengths.

The command will fail if the CAN core already is in reset mode.

14.5.7.3. GET_STATS

This call copies the driver's internal counters to an user provided data area. The format of the data written is described in the data structure subsection. See the *occan_stats* data structure.

The call will fail if the pointer to the data is invalid and errno will be set to EINVAL.

14.5.7.4. GET_STATUS

This call stores the current status of the CAN core to the address pointed to by the argument given to *ioctl*. This call is typically used to determine the error state of the CAN core. The 4 byte status bit mask can be interpreted as in the table below.

Table 14.9. Status bit mask.

Mask	Description
OCCAN_STATUS_RESET	Core is in reset mode
OCCAN_STATUS_OVERRUN	Data overrun
OCCAN_STATUS_WARN	Has passed the error warning limit (96)
OCCAN_STATUS_ERR_PASSIVE	Has passed the Error Passive limit (127)
OCCAN_STATUS_ERR_BUSOFF	Core is in reset mode due to a bus off (255)

This call never fail.

14.5.7.5. SET_SPEED

The SET_SPEED ioctl call is used to set the baud rate of the CAN bus. The timing register values are calculated for the given baud rate. The baud rate is given in Hertz. For the baud rate calculations to function properly the driver needs to know it's frequency, if application is started from GRMON make sure that the correct frequency is detected or that the -freq option is being used. This command calculates the timing registers from the core frequency, the sampling point location and the baud rate given here. For custom timing parameters, please use SET_BTRS instead.

If the calculated timing parameters result in a baud rate that are worse than 10% wrong, errno will be set to EINVAL and the return code is -1.

14.5.7.6. SET_BTRS

This call sets the timing registers manually. See the CAN hardware documentation for a detailed description of the timing parameters. The SET_BTRS call takes a 32-bit integer argument containing all available timing parameters. The lower 7..0 bits is BTR1 and 15..8 bits is BTR0. It is encouraged to use this function over the SET_SPEED.

This call fail if the CAN core is in running mode, in that case errno will be set to EBUSY and ioctl will return -1.

14.5.7.7. SET_BLK_MODE

This call sets blocking mode for receive and transmit operations, i.e. read and write. Input is a bit mask as described in the table below.

Table 14.10. SET_BLK_MODE ioctl argument

Bit number	Description
OCCAN_BLK_MODE_RX	Set this bit to make read block when no messages can be read.
OCCAN_BLK_MODE_TX	Set this bit to make *write* block until all messages has been sent or put info software fifo.

This call never fails, it is valid to call this command in any mode.

14.5.7.8. SET_BUFLEN

This call sets the buffer length of the receive and transmit software FIFOs. To set the FIFO length the core needs to be in reset mode. In the table below the input to the ioctl command is described.

Table 14.11. SET_BUFLEN ioctl argument

Mask	Description
0x0000ffff	Receive buffer length in number of *CANMsg* structures.
0xffff0000	Transmit buffer length in number of *CANMsg* structures.

Errno will be set to ENOMEM when the driver was not able to get the requested memory amount. EBUSY is set when the core is in operating mode.

14.5.7.9. SET_FILTER

Set Acceptance filter matched by receiver for every message that is received. Let the second argument point to a occan_afilter data structure or NULL to disable filtering to let all messages pass the filter.

14.5.8. Transmission

Transmitting messages are done with the write call. It is possible to write multiple messages in one call. An example of a write call is shown below:

```
result = write(fd, &tx_msgs[0], sizeof(CANMsg)*msgcnt))
```

On success the number of transmitted bytes is returned and -1 on failure. Errno is also set in the latter case. Tx_msgs points to the beginning of the CANMsg structure which includes id, type of message, data and data length. The last parameter sets the number of CAN messages that will be transmitted it must be a multiple of CANMsg structure size.

The call will fail if the user tries to send more bytes than is allocated for a single message (this can be changed with the SET_PACKETSIZE ioctl call) or if a NULL pointer is passed.

The write call can be configured to block when the software FIFO is full. In non-blocking mode, write will immediately return either return -1 indicating that no messages were written or the total number of bytes written (always a multiple of CANMsg structure size). Note that a 3 message write request, may end up in only 2 written, the caller is responsible to check the number of messages actually written in non-blocking mode.

If no resources are available in non-blocking mode the call will return with an error. The errno variable is set according to the table given below.

Table 14.12. Write errno values.

ERRNO	Description
EINVAL	An invalid argument was passed. For example The buffer length was less than a single CANMsg structure size, or NULL pointer.
EBUSY	The link is not in operating mode, but in reset mode. Nothing done.
EWOULDBLOCK	Returned only in non-blocking mode, the requested operation failed due to no transmission buffers available, the driver would have blocked the calling task in blocking mode. No buffers available for transmission, no message were scheduled for transmission.
EIO	Calling task was woken up from blocking mode by a bus off error. The CAN core has entered reset mode. Further calls to read or write will fail until the *ioctl* command START is issued again.

Each Message has an individual set of options controlled in the CANMsg structure. See the data structure subsection for structure member descriptions.

14.5.9. Reception

Reception of CAN messages from the CAN bus can be done using the read call. An example is shown below:

```
CANMsg rx_msgs[5];
len = read(fd, rx_msgs, 5 * sizeof(rx_msgs));
```

The requested number of bytes to be read is given in the third argument. The messages will be stored in rx_msgs. The actual number of received bytes (a multiple of sizeof(CANMsg)) is returned by the function on success and -1 on failure. In the latter case *errno* is also set.

The CANMsg data structure is described in the data structure subsection.

The call will fail if a NULL pointer is passed, invalid buffer length, the CAN core is in stopped mode or due to a bus off error while still in the read function.

The blocking behavior can be set using ioctl calls. In blocking mode the call will block until at least one message has been received. In non-blocking mode, the call will return immediately and if no message was available -1 is returned and errno set appropriately. The table below shows the different errno values returned.

Table 14.13. Read errno values.

ERRNO	**Description**
EINVAL	A NULL pointer was passed as the data pointer or the length was illegal.
EBUSY	CAN core is in reset mode. Switch to operating mode by issuing a START ioctl command.
EWOULDBLOCK	In non-blocking mode, no messages were available in the software receive FIFO. In blocking mode the driver would block the calling task until at least one new messages have been received.
EIO	A blocking read was interrupted due to the the CAN core leaved operating mode and entered reset mode, this might be due to a bus off error or the device was closed. Further calls to read or write will fail until the ioctl command START is issued again, or reopened.

15. AHB Status register driver

15.1. Overview

This section describes the GRLIB AHBSTAT device driver that is available in the SPARC/VxWorks 6.7 distribution.

15.2. Show routines

During debugging it is sometimes useful to see that the hardware have been discovered correctly by the driver. The AHBSTAT driver provides two functions listed below that can be used directly from the VxWorks C shell or in an application.

```
void ahbstatShow(void)
void ahbstatShowMinor(int minor)
```

15.3. Source code

Sources are located at `vxworks-6.7/target/src/hwif/grlib/grlibAhbstat.c` and the interface declaration is included from `hwif/grlib/grlibAhbstat.h`. The driver is included by adding the component

15.4. Operation

The AHBSTAT device can generate an interrupt on AHB error responses and other access errors (correctable errors for example) by looking at signals alongside the AMBA bus. The signals are generated from FT-cores at the time the correctable AMBA access takes place. The software may through interrupt and by looking at the AHBSTAT registers determine what type of error happended and the address accessed.

This driver initializes the AHBSTAT and enables interrupt handling on initialization. A default interrupt handler is installed which prints detected access error to the system console through `logMsg()`, and the AHBSTAT is reenabled by software in order to detect the next error. The user may override the default interrupt handler to do custom handling and some function exists to make accesses to the AHBSTAT easier.

15.5. User interface

The driver provides the ability to assign a custom interrupt error handler and through a simple function interface read the last error that occured which as been sampled at the last interrupt. The interrupt handler will be called first after the "InstConnect" phase of the vxBus framework initialization. If AHBSTAT errors are to be detected earlier than that, one must poll the register content without relying on interrupt generation. The AHBSTAT is initialized to detect errors during the "InstConnect1" phase of the VxBus framework.

15.5.1. Assigning a custom interrupt handler

A custom interrupt handlers can be installed to handle events as a result of AMBA errors detected. The AHBSTAT driver has a weak function pointer which can be overrided at compile time or at runtime. Below is the prototype for the IRQ handler (ISR) install function.

```
int (*ahbstat_error)(
  int minor,
  struct ahbstat_regs *regs,
  unsigned long status,
  unsigned long failing_address
)
```

The function is called from the AHBSTAT Interrupt Service Routine (ISR), the AHBSTAT driver will provide the custom function with four arguments described in the table below. The return value is interpreted as a 2-bit bit-mask, where bit zero turns on or off the default printout and the second bit controls wheather the AHBSTAT is to be reenabled or not.

Table 15.1. ahbstat_error ISR handler argument description

Argument Name	Description
minor	AHBSTAT device index
regs	Base address of AHBSTAT registers
status	STATUS register sampled by the AHBSTAT driver ISR. This tells the custom interrupt handler what error just occured.
failing_address	FAILING ADDRESS register sampled by the AHBSTAT driver ISR. This tells the custom interrupt handler at what address the error occured.

15.5.2. Get the last AHB error occured

The AHBSTAT driver records the last error into a temporary variable in memory on every error interrupt. The value may be read from the function `ahbstat_last_error`, the prototype is listed below. The argument [minor] determines which AHBSTAT device to request the information from. The second ([status]) and third ([address]) arguments are pointers to memory where the function will store the recorded information to.

```
int ahbstat_last_error(
    int minor,
    unsigned long *status,
    unsigned long *address)
```

15.5.3. Reenable the AHB error detection and IRQ generation

To force a reenabling of the AHB error detection the `ahbstat_reset` can be called, the prototype is listed below. The argument [minor] determines which AHBSTAT device to reenable. By defualt the Interrupt handler will reenable the AHB error detection.

```
void ahbstat_reset(int minor)
```

15.5.4. AHBSTAT device registers

The registers can be accessed directly in order to support custom handling. The base address of the registers are returned by the `ahbstat_get_regs` function. The argument [minor] determines which AHBSTAT device. If no AHBSTAT matching the [minor] number is found, NULL is returned.

```
struct ahbstat_regs *ahbstat_get_regs(int minor)
```

16. GR1553B Driver

16.1. Introduction

Documentation for the GR1553B driver. See known limitations below.

This document describes the VxWorks 6.7 drivers specific to the GRLIB GR1553B core. The Remote Terminal(RT), Bus Monitor (BM and Bus Controller (BC) functionality are supported by the driver. Device discovery and resource sharing are commonly controlled by the GR1553B driver described in this chapter. Each 1553 mode is supported by a separate driver, the drivers are documented in separate chapters.

All the GR1553B drivers relies on the WindRiver VxBus framework for the services: device detection, driver loading/initialization and interrupt management. VxBus is responsible for creating GR1553B device instances and uniting GR1553B devices with the GR1553B low-level driver. For reference, documentation about the VxBus framework is found in WindRiver documentation that comes with VxWorks.

This section gives an brief introduction to the GRLIB GR1553B device allocation driver used internally by the BC, BM and RT device drivers. This driver controls the GR1553B device regardless of interfaces supported (BC, RT and/or BM). The device can be located at an on-chip AMBA or an AMBA-over-PCI bus. The driver provides an interface for the BC, RT and BM drivers.

Since the different interfaces (BC, BM and RT) are accessed from the same register interface on one core, the APB device must be shared among the BC, BM and RT drivers. The GR1553B driver provides an easy function interface that allows the APB device to be shared safely between the BC, BM and RT device drivers.

Any combination of interface functionality is supported, but the RT and BC functionality cannot be used simultaneously (limited by hardware).

The interface towards to the BC, BM and RT drivers is used internally by the device drivers and is not documented here. See respective driver for an interface description.

16.1.1. Considerations and limitations

Note that the following items must be taken into consideration when using the GR1553B drivers:

- The driver uses only Physical addressing, i.e it does not do MMU translation or memory mapping for the user. The user is responsible for mapping DMA memory buffers provided to the 1553 drivers 1:1.
- Physical buffers addresses (assigned by user) must be located at non-cachable areas or D-Cache snooping must be present in hardware. If D-cache snooping is not present the user must edit the GR1553*_READ_MEM() macros in respective driver.
- SMP locking (spin-locks) has not been implemented, it does however not mean that SMP mode can not be used. The CPU handling the IRQ (CPU0 unless configured otherwise) must be the CPU and only CPU using the driver API. Only one CPU can use respective driver API at a time. One can use the CPU affinity ability of the VxWorks Scheduler to configure that only one CPU shall eecute a specific task.

The above restrictions should not cause any problems for the AT697 + GR-RASTA-IO (RASTA-101) systems or similar.

16.1.2. GR1553B Hardware

The GRLIB GR1553B core may support up to three modes depending on configuration, Bus Controller (BC), Remote Terminal (RT) or Bus Monitor (BM). The BC and RT functionality may not be used simultaneously, but the BM may be used together with BC or RT or separately. All three modes are supported by the driver.

Interrupts generated from BC, BM and RT result in the same system interrupt, interrupts are shared.

16.1.3. Software Driver

The driver provides an interface used internally by the BC, BM and RT device drivers, see respective driver for an interface declaration. The driver sources and definitions are listed in the table below, the path is given relative to the VxWorks source tree `vxworks-6.7/target`.

Table 16.1. Source Location

Filename	Description
src/hwif/grlib/gr1553b.c	GR1553B Driver source
h/hwif/grlib/gr1553b.h	GR1553B Driver interface declaration

16.1.4. Driver Registration

The driver must be registered to the VxBus framework. The registration is performed by including the GR1553B Component named DRV_GRLIB_GR1553B. This driver is automatically registered when including any of the BC, BM and the RT device drivers.

The registration of the driver is crucial for the user to be able to access the driver application programming interfaces. The drivers does not use the VxWorks I/O layer, instead a classic C-language API is used from kernel mode tasks.

The driver is called from the VxBus infrastructure to handle detected GR1553B hardware. In order for VxBus to match the GR1553B driver with the GR1553B hardware device one must register the driver into the VxBus framework. This process is automatically done when including the driver into the project, which can be done graphically from the WindRiver Workbench. The drivers are added independently into the project by including/defining one or multiple of the components listed below.

- DRV_GRLIB_GR1553B required by all GR1553B drivers (described in this chapter)
- DRV_GRLIB_GR1553BC is the Bus Controller driver
- DRV_GRLIB_GR1553RT is Remote Terminal driver
- DRV_GRLIB_GR1553B is the Bus Monitor driver

16.1.5. Examples

Examples of how to use the VxWorks GR1553 driver is not currently available. The Aeroflex Gaisler RTEMS RCC distribution contains some examples which also includes the GR1533B driver providing the same API to users. VxWorks examples can be received on request to support@gaisler.com, the status is however not fully tested and has therefore not included in the release.

17. GR1553B Bus Controller Driver

17.1. Introduction

This section describes the GRLIB GR1553B Bus Controller (BC) device driver interface. The driver relies on the GR1553B driver and the VxBus. The reader is assumed to be well acquainted with MIL-STD-1553 and the GR1553B core. The GR1553BC device driver is included by adding the DRV_GRLIB_GR1553BC component.

17.1.1. GR1553B Bus Controller Hardware

The GR1553B core supports any combination of the Bus Controller (BC), Bus Monitor (BM) and Remote Terminal (RT) functionality. This driver supports the BC functionality of the hardware, it can be used simultaneously with the Bus Monitor (BM) functionality. When the BM is used together with the BC, interrupts are shared between the drivers.

The three functions (BC, BM, RT) are accessed using the same register interface, but through separate registers. In order to share hardware resources between the three GR1553B drivers, the three depends on a lower level GR1553B driver, see Chapter 16.

The driver supports the on-chip AMBA bus and the AMBA-over-PCI bus.

17.1.2. Software Driver

The BC driver is split in two parts, one where the driver access the hardware device and one part where the descriptors are managed. The two parts are described in two separate sections below.

Transfer and conditional descriptors are collected into a descriptor list. A descriptor list consists of a set of Major Frames, which consist of a set of Minor Frames which in turn consists of up to 32 descriptors (also called Slots). The composition of Major/Minor Frames and slots is configured by the user, and is highly dependent of application.

The Major/Minor/Slot construction can be seen as a tree, the tree does not have to be symmetrically, i.e. Major frames may contain different numbers of Minor Frames and Minor Frames may contain different numbers of Slot.

GR1553B BC descriptor lists are generated by the list API available in `gr1553bc_list.h`.

The driver provides the following services:

- Start, Stop, Pause and Resume descriptor list execution
- Synchronous and asynchronous descriptor list management
- Interrupt handling
- BC status
- Major/Minor Frame and Slot (descriptor) model of communication
- Current Descriptor (Major/Minor/Slot) Execution Indication
- Software External Trigger generation, used mainly for debugging or custom time synchronization
- Major/Minor Frame and Slot/Message ID
- Minor Frame time slot management

The driver sources and definitions are listed in the table below, the path is given relative to the VxWorks source tree `vxworks-6.7/target`.

Table 17.1. BC driver Source location

Filename	Description
src/hwif/grlib/gr1553bc.c	GR1553B BC Driver source
src/hwif/grlib/gr1553bc_list.c	GR1553B BC List handling source
h/hwif/grlib/gr1553bc.h	GR1553B BC Driver interface declaration
h/hwif/grlib/gr1553bc_list.h	GR1553B BC List handling interface declaration

17.2. BC Device Handling

The BC device driver's main purpose is to start, stop, pause and resume the execution of descriptor lists. Lists are described in the Descriptor List section 1.3 . In this section services related to direct access of BC hardware registers and Interrupt are described. The function API is declared in `gr1553bc.h`.

17.2.1. Device API

The device API consists of the functions in the table below.

Table 17.2. Device API function prototyper

Prototype	Description
`void *gr1553bc_open(int minor)`	Open a BC device by minor number. Private handle returned used in all other device API functions.
`void gr1553bc_close(void *bc)`	Close a previous opened BC device.
`int gr1553bc_start(void *bc, struct gr1553bc_list *list, struct gr1553bc_list *list_async)`	Schedule a synchronous and/or a asynchronous BC descriptor Lists for execution. This will unmask BC interrupts and start executing the first descriptor in respective List. This function can be called multiple times.
`int gr1553bc_pause(void *bc)`	Pause the synchronous List execution.
`int gr1553bc_restart(void *bc)`	Restart the synchronous List execution.
`int gr1553bc_stop(void *bc, int options)`	Stop Synchronous and/or asynchronous list.
`int gr1553bc_indication(void *bc, int async, int *mid)`	Get the current BC hardware execution position (MID) of the synchronous or asynchronous list.
`void gr1553bc_status(void *bc, struct gr1553bc_status *status)`	GGet the BC hardware status and time.
`void gr1553bc_ext_trig(void *bc, int trig)`	Trigger an external trigger by writing to the BC action register.
`int gr1553bc_irq_setup(void *bc, bcirq_func_t func, void *data)`	Generic interrupt handler configuration. Handler will be called in interrupt context on errors and interrupts generated by transfer descriptors.

17.2.1.1. Data Structures

The `gr1553bc_status` data structure contains the BC hardware status sampled by the function `gr1553bc_status()`.

```
struct gr1553bc_status {
    unsigned int status;
    unsigned int time;
};
```

Table 17.3. gr1553bc_status member descriptions

Member	Description
status	BC status register

Member	Description
time	BC Timer register

17.2.1.2. gr1553bc_open

Opens a GR1553B BC device by device instance index. The minor number relates to the order in which a GR1553B BC device is found in the Plug&Play information. A GR1553B core which lacks BC functionality does not affect the minor number.

If a BC device is successfully opened a pointer is returned. The pointer is used internally by the GR1553B BC driver, it is used as the input parameter *bc* to all other device API functions.

If the driver failed to open the device, NULL is returned.

17.2.1.3. gr1553bc_close

Closes a previously opened BC device. This action will stop the BC hardware from processing descriptors/ lists, disable BC interrupts, and free dynamically memory allocated by the driver.

17.2.1.4. gr1553bc_start

Calling this function starts the BC execution of the synchronous list and/or the asynchronous list. At least one list pointer must be non-zero to affect BC operation. The BC communication is enabled depends on list, and Interrupts are enabled.

This function can be called multiple times. If a list (of the same type) is already executing it will be replaced with the new list.

17.2.1.5. gr1553bc_pause

Pause the synchronous list. It may be resumed by `gr1553bc_resume()`. See hardware documentation.

17.2.1.6. gr1553bc_resume

Resume the synchronous list, must have been previously paused by `gr1553bc_pause()`. See hardware documentation.

17.2.1.7. gr1553bc_stop

Stop synchronous and/or asynchronous list execution. The second argument is a 2-bit bit-mask which determines the lists to stop, see table below for a description.

Table 17.4. gr1553bc_stop second argument

Member	Description
Bit 0	Set to one to stop the synchronous list.
Bit 1	Set to one to stop the asynchronous list.

17.2.1.8. gr1553bc_indication

Retrieves the current Major/Minor/Slot (MID) position executing into the location indicated by *mid*. The async argument determines which type of list is queried, the Synchronous (*async*=0) list or the Asynchronous (*async*=1).

Note that since the List API internally adds descriptors the indication may seem to be out of bounds.

17.2.1.9. gr1553bc_status

This function retrieves the current BC hardware status. Second argument determine where the hardware status is stored, the layout of the data stored follows the `gr1553bc_status` data structure. The data structure is described in Table 17.3.

17.2.1.10. gr1553bc_ext_trig

The BC supports an external trigger signal input which can be used to synchronize 1553 transfers. If used, the external trigger is normally generated by some kind of Time Master. A message slot may be programmed to wait for an external trigger before being executed, this feature allows the user to accurate send time synchronize messages to RTs. However, during debugging or when software needs to control the time synchronization behaviour the external trigger pulse can be generated from the BC core itself by writing the BC Action register.

This function sets the external trigger memory to one by writing the BC action register.

17.2.1.11. gr1553bc_irq_setup

Install a generic handler for BC device interrupts. The handler will be called on Errors (DMA errors etc.) resulting in interrupts or transfer descriptors resulting in interrupts. The handler is not called when an IRQ is generated by a condition descriptor. Condition descriptors have their own custom handler.

Condition descriptors are inserted into the list by user, each condition may have a custom function and data assigned to it, see `gr1553bc_slot_irq_prepare()`. Interrupts generated by condition descriptors are not handled by this function.

The third argument is custom data which will be given to the handler on interrupt.

17.3. Descriptor List Handling

The BC device driver can schedule synchronous and asynchronous lists of descriptors. The list contains a descriptor table and a software description to make certain operations possible, for example translate descriptor address into descriptor number (MID).

The BC stops execution of a list when a END-OF-LIST (EOL) marker is found. Lists may be configured to jump to the start of the list (the first descriptor) by inserting an unconditional jump descriptor. Once a descriptor list is setup the hardware may process the list without the need of software intervention. Time distribution may also be handled completely in hardware, by setting the "Wait for External Trigger" flag in a transfer descriptor the BC will wait until the external trigger is received or proceed directly if already received. See hardware manual.

17.3.1. Overview

This section describes the Descriptor List Application Programming Interface (API). It provides functionality to create and manage BC descriptor lists.

A list is built up by the following building blocks:

- Major Frame (Consists of N Minor Frames)
- Minor Frame (Consists of up to 32 1553 Slots)
- Slot (Transfer/Condition BC descriptor), also called Message Slot

The user can configure lists with different number of Major Frames, Minor Frames and slots within a Minor Frame. The List manages a strait descriptor table and a Major/Minor/Slot tree in order to easily find it's way through all descriptor created.

Each Minor frame consist of up to 32 slot and two extra slots for time management and descriptor find operations, see figure below. In the figure there are three Minor frames with three different number of slots 32, 8 and 4. The List manage time slot allocation per Minor frame, for example a minor frame may be programmed to take 8ms and when the user allocate a message slot within that Minor frame the time specified will be subtracted from the 8ms, and when the message slot is freed the time will be returned to the Minor frame again.

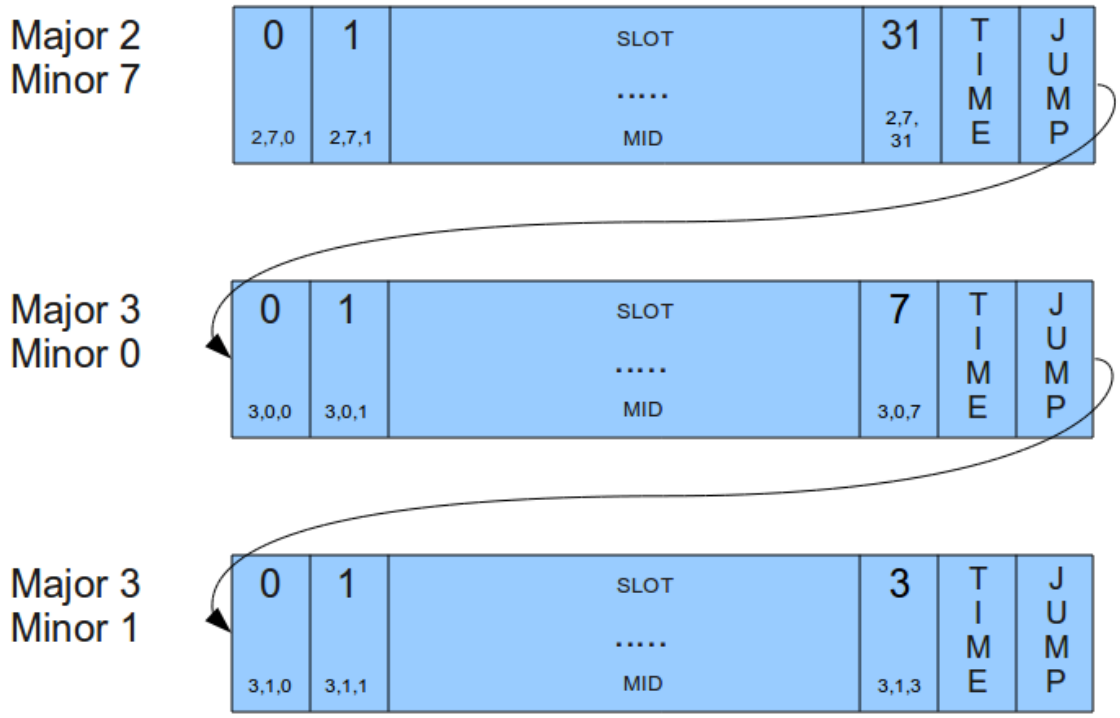

Figure 17.1. Three consecutive Minor Frames

A specific Slot [Major, Minor, Slot] is identified using a MID (Message-ID). The MID consist of three numbers Major Frame number, Minor Frame number and Slot Number. The MID is a way for the user to avoid using descriptor pointers to talk with the list API. For example a condition Slot that should jump to a message Slot can be created by knowing "MID and Jump-To-MID". When allocating a Slot (with or without time) in a List the user may specify a certain Slot or a Minor frame, when a Minor frame is given then the API will find the first free Slot as early in the Minor Frame as possible and return it to the user.

A MID can also be used to identify a certain Major Frame by setting the Minor Frame and Slot number to 0xff. A Minor Frame can be identified by setting Slot Number to 0xff.

A MID can be created using the macros in the table below.

Table 17.5. Macros for creating MID

MACRO Name	**Description**
GR1553BC_ID(major,minor,slot)	ID of a SLOT
GR1553BC_MINOR_ID(major,minor)	ID of a MINOR (Slot=0xff)
GR1553BC_MAJOR_ID(major)	ID of a Major (Minor=0xff,Slot=0xff)

17.3.2. Example: steps for creating a list

The typical approach when creating lists and executing it:

- gr1553bc_list_alloc(&list, MAJOR_CNT)
- gr1553bc_list_config(list, &listcfg)
- Create all Major Frames and Minor frame, for each major frame:
  1. gr1553bc_major_alloc_skel(&major, &major_minor_cfg)
  2. gr1553bc_list_set_major(list, &major, MAJOR_NUM)

- Link last and first Major Frames together:

  1. gr1553bc_list_set_major(&major7, &major0)

- gr1553bc_list_table_alloc() (Allocate Descriptor Table)
- gr1553bc_list_table_build() (Build Descriptor Table from Majors/Minors)
- Allocate and initialize Descriptors pre defined before starting:

  1. gr1553bc_slot_alloc(list, &MID, TIME_REQUIRED, ..)
  2. gr1553bc_slot_transfer(MID, ..)

- START BC HARDWARE BY SCHDULING ABOVE LIST
- Application operate on executing List

17.3.3. Major Frame

Consists of multiple Minor frames. A Major frame may be connected/linked with another Major frame, this will result in a Jump Slot from last Minor frame in the first Major to the first Minor in the second Major.

17.3.4. Minor Frame

Consists of up to 32 Message Slots. The services available for Minor Frames are Time-Management and Slot allocation.

Time-Management is optional and can be enabled per Minor frame. A Minor frame can be assigned a time in microseconds. The BC will not continue to the next Minor frame until the time specified has passed, the time includes the 1553 bus transfers. See the BC hardware documentation. Time is managed by adding an extra Dummy Message Slot with the time assigned to the Minor Frame. Every time a message Slot is allocated (with a certain time: Slot-Time) the Slot-Time will be subtracted from the assigned time of the Minor Frame's Dummy Message Slot. Thus, the sum of the Message Slots will always sum up to the assigned time of the Minor Frame, as configured by the user. When a Message Slot is freed, the Dummy Message Slot's Slot-Time is incremented with the freed Slot-Time. See figure below for an example where 6 Message Slots has been allocated Slot-Time in a 1 ms Time-Managed Minor Frame. Note that in the example the Slot-Time for Slot 2 is set to zero in order for Slot 3 to execute directly after Slot 2.

Figure 17.2. Time-Managed Minor Frame of 1ms

The total time of all Minor Frames in a Major Frame determines how long time the Major Frame is to be executed.

Slot allocation can be performed in two ways. A Message Slot can be allocated by identifying a specific free Slot (MID identifies a Slot) or by letting the API allocate the first free Slot in the Minor Frame (MID identifies a Minor Frame by setting Slot-ID to 0xff).

17.3.5. Slot (Descriptor)

The GR1553B BC core supports two Slot (Descriptor) Types:

- Transfer descriptor (also called Message Slot)

- Condition descriptor (Jump, unconditional-IRQ)

See the hardware manual for a detail description of a descriptor (Slot).

The BC Core is unaware of lists, it steps through executing each descriptor as the encountered, in a sequential order. Conditions resulting in jumps gives the user the ability to create more complex arrangements of buffer descriptors (BD) which is called lists here.

Transfer Descriptors (TBD) may have a time slot assigned, the BC core will wait until the time has expired before executing the next descriptor. Time slots are managed by Minor frames in the list. See Minor Frame section. A Message Slot generating a data transmission on the 1553 bus must have a valid data pointer, pointing to a location from which the BC will read or write data.

A Slot is allocated using the `gr1553bc_slot_alloc()` function, and configured by calling one of the function described in the table below. A Slot may be reconfigured later. Note that a conditional descriptor does not have a time slot, allocating a time for a conditional times slot will lead to an incorrect total time of the Minor Frame.

Table 17.6. Slot configuration

Function Name	Description
gr1553bc_slot_irq_prepare	Unconditional IRQ slot
gr1553bc_slot_jump	Unconditional jump
gr1553bc_slot_exttrig	Dummy transfer, wait for EXTERNAL-TRIGGER
gr1553bc_slot_transfer	Transfer descriptor
gr1553bc_slot_empty	Create Dummy Transfer descriptor
gr1553bc_slot_raw	Custom Descriptor handling

Existing configured Slots can be manipulated with the following functions.

Table 17.7. Slot manipulation

Function Name	Description
gr1553bc_slot_dummy	Set existing Transfer descriptor to Dummy. No 1553 bus transfer will be performed.
gr1553bc_slot_update	Update Data Pointer and/or Status of a TBD

17.3.6. Changing a scheduled BC list (during BC-runtime)

Changing a descriptor that is being executed by the BC may result in a race between hardware and software. One of the problems is that a descriptor contains multiple words, which can not be written simultaneously by the CPU. To avoid the problem one can use the INDICATION service to avoid modifying a descriptor currently in use by the BC core. The indication service tells the user which Major/Minor/ Slot is currently being executed by hardware, from that information an knowing the list layout and time slots the user may safely select which slot to modify or wait until hardware is finished.

In most cases one can do descriptor initialization in several steps to avoid race conditions. By initializing (allocating and configuring) a Slot before starting the execution of the list, one may change parts of the descriptor which are ignored by the hardware. Below is an example approach that will avoid potential races between software and hardware:

1. Initialize Descriptor as Dummy and allocated time (often done before starting/ scheduling list)
2. The list is started, as a result descriptors in the list are executed by the BC
3. Modify transfer options and data-pointers, but maintain the Dummy bit.
4. Clear the Dummy bit in one atomic data store.

17.3.7. Custom Memory Setup

For designs where dynamically memory is not an option, or the driver is used on an AMBA-over-PCI bus (where `malloc()` does not work), the API allows the user to provide custom addresses for the descriptor table and object descriptions (lists, major frames, minor frames).

Being able to configure a custom descriptor table may for example be used to save space or put the descriptor table in on-chip memory. The descriptor table is setup using the function `gr1553bc_list_table_alloc(list, CUSTOM_ADDRESS)`.

Object descriptions are normally allocated during initialization procedure by providing the API with an object configuration, for example a Major Frame configuration enables the API to dynamically allocate the software description of the Major Frame and with all it's Minor frames. Custom object allocation requires internal understanding of the List management parts of the driver, it is not described in this document.

17.3.8. Interrupt handling

There are different types of interrupts, Error IRQs, transfer IRQs and conditional IRQs. Error and transfer Interrupts are handled by the general callback function of the device driver. Conditional descriptors that cause Interrupts may be associated with a custom interrupt routine and argument.

Transfer Descriptors can be programmed to generate interrupt, and condition descriptors can be programmed to generate interrupt unconditionally (there exists other conditional types as well). When a Transfer descriptor causes interrupt the general ISR callback of the BC driver is called to let the user handle the interrupt. Transfers descriptor IRQ is enabled by configuring the descriptor.

When a condition descriptor causes an interrupt a custom IRQ handler is called (if assigned) with a custom argument and the descriptor address. The descriptor address my be used to lookup the MID of the descriptor. The API provides functions for placing unconditional IRQ points anywhere in the list. Below is an pseudo example of adding an unconditional IRQ point to a list:

```
void funcSetup()
{
    int MID;

    /* Allocate Slot for IRQ Point */
    gr1553bc_slot_alloc(&MID, TIME=0, ..);

    /* Prepare unconditional IRQ at allocated SLOT */
    gr1553bc_slot_irq_prepare(MID, funcISR, data);

    /* Enabling the IRQ may be done later during list
     * execution */
    gr1553bc_slot_irq_enable(MID);
}
void funcISR(*bd, *data)
{
    /* HANDLE ONE OR MULTIPLE DESCRIPTORS
     *(MULTIPLE IN THIS EXAMPLE): */
    int MID;

    /* Lookup MID from descriptor address */
    gr1553bc_mid_from_bd(bd, &MID, NULL);

    /* Print MID which caused the Interrupt */
    printk("IRQ ON %06x\n", MID);
}
```

17.3.9. List API

Table 17.8. List API function prototypes

Prototype	Description
`int gr1553bc_list_alloc( struct gr1553bc_list **list, int max_major)`	Allocate a List description structure. First step in creating a descriptor list.
`void gr1553bc_list_free( struct gr1553bc_list *list)`	Free a List previously allocated using `gr1553bc_list_alloc()`.

Prototype	Description

`int gr1553bc_list_config(`
`  struct gr1553bc_list *list,`
`  struct gr1553bc_list_cfg *cfg,`
`  void *bc)`
	Configure List parameters and associate it with a BC device that will execute the list later on. List parameters are used when generating descriptors.

`void gr1553bc_list_link_major(`
`  struct gr1553bc_major *major,`
`  struct gr1553bc_major *next)`
	Links two Major frames together, the Major frame indicated by next will be executed after the Major frame indicated by major. A unconditional jump is inserted to implement the linking.

`int gr1553bc_list_set_major(`
`  struct gr1553bc_list *list,`
`  struct gr1553bc_major *major,`
`  int no)`
	Assign a Major Frame a Major Frame number in a list. This will link Major (no-1) and Major (no+1) with the Major frame, the linking can be changed by calling `gr1553bc_list_link_major()` after all major frames have been assigned a number.

`int gr1553bc_minor_table_size(` `  struct gr1553bc_minor *minor)`	Calculate the size required in the descriptor table by one minor frame.
`int gr1553bc_list_table_size(` `  struct gr1553bc_list *list)`	Calculate the size required for the complete descriptor list.

`int gr1553bc_list_table_alloc(`
`  struct gr1553bc_list *list,`
`  void *bdtab_custom)`
	Allocate and initialize a descriptor list. The `bdtab_custom` argument can be used to assign a custom address of the descriptor list.

`void gr1553bc_list_table_free(` `  struct gr1553bc_list *list)`	Free descriptor list memory previously allocated by `gr1553bc_list_table_alloc()`.
`int gr1553bc_list_table_build(` `  struct gr1553bc_list *list)`	Build all descriptors in a descriptor list. Unused descriptors will be initialized as empty dummy descriptors. After this call descriptors can be initialized by user.

`int gr1553bc_major_alloc_skel(`
`  struct gr1553bc_major **major,`
`  struct gr1553bc_major_cfg *cfg)`
	Allocate and initialize a software description skeleton of a Major Frame and it's Minor Frames.

`int gr1553bc_list_freetime(`
`  struct gr1553bc_list *list,`
`  int mid)`
	Get total unused slot time of a Minor Frame. Only available if time management has been enabled for the Minor Frame.

`int gr1553bc_slot_alloc(`
`  struct gr1553bc_list *list,`
`  int *mid,`
`  int timeslot,`
`  union gr1553bc_bd **bd)`
	Allocate a Slot from a Minor Frame. The Slot location is identified by MID. If the MID identifies a Minor frame the first free slot is allocated within the minor frame.

`int gr1553bc_slot_free(`
`  struct gr1553bc_list *list,`
`  int mid)`
	Return a previously allocated Slot to a Minor Frame. The slot-time is also returned.

`int gr1553bc_mid_from_bd(`
`  union gr1553bc_bd *bd,`
`  int *mid,`
`  int *async)`
	Get Slot/Message ID from descriptor address.

`union gr1553bc_bd *gr1553bc_slot_bd(`
`  struct gr1553bc_list *list,`
`  int mid)`
	Get descriptor address from MID.

`int gr1553bc_slot_irq_prepare(`
`  struct gr1553bc_list *list,`
`  int mid,`
`  bcirq_func_t func,`
`  void *data)`
	Prepare a condition Slot for generating interrupt. Interrupt is disabled. A custom callback function and data is assigned to Slot.

`int gr1553bc_slot_irq_enable(`
`  struct gr1553bc_list *list,`
`  int mid)`
	Enable interrupt of a previously interrupt-prepared Slot.

`int gr1553bc_slot_irq_disable(`
`  struct gr1553bc_list *list,`
`  int mid)`
	Disable interrupt of a previously interrupt-prepared Slot.

`int gr1553bc_slot_jump(`
`  struct gr1553bc_list *list,`
`  int mid,`
`  uint32_t condition,`
`  int to_mid)`
	Initialize an allocated Slot, the descriptor is initialized as a conditional Jump Slot. The conditional is controlled by the third argument. The Slot jumped to is determined by the fourth argument.

Prototype	Description
`int gr1553bc_slot_exttrig( struct gr1553bc_list *list, int mid)`	Create a dummy transfer with the "Wait for external trigger" bit set.
`int gr1553bc_slot_transfer( struct gr1553bc_list *list, int mid, int options, int tt, uint16_t *dptr)`	Create a transfer descriptor.
`int gr1553bc_slot_dummy( struct gr1553bc_list *list, int mid, unsigned int *dummy)`	Manipulate the DUMMY bit of a transfer descriptor. Can be used to enable or disable a transfer descriptor.
`int gr1553bc_slot_empty( struct gr1553bc_list *list, int mid)`	Create an empty transfer descriptor, with the DUMMY bit set. The time- slot previously allocated is preserved.
`int gr1553bc_slot_update( struct gr1553bc_list *list, int mid, uint16_t *dptr, unsigned int *stat)`	Update a transfer descriptors data pointer and/or status field.
`int gr1553bc_slot_raw( struct gr1553bc_list *list, int mid, unsigned int flags, uint32_t word0, uint32_t word1, uint32_t word2, uint32_t word3)`	Custom descriptor initialization. Note that a bad initialization may break the BC driver.
`void gr1553bc_show_list( struct gr1553bc_list *list, int options)`	Print information about a descriptor list to standard out. Used for debugging.

17.3.9.1. Data structures

The `gr1553bc_major_cfg` data structure hold the configuration parameters of a Major frame and all it's Minor frames. The `gr1553bc_minor_cfg` data structure contain the configuration parameters of one Minor Frame.

```
struct gr1553bc_minor_cfg {
    int slot_cnt;
    int timeslot;
};

struct gr1553bc_major_cfg {
    int minor_cnt;
    struct gr1553bc_minor_cfg minor_cfgs[1];
};
```

Table 17.9. gr1553bc_minor_cfg member descriptions.

Member	Description
slot_cnt	Number of Slots in Minor Frame
timeslot	Total time-slot of Minor Frame [us]

Table 17.10. gr1553bc_major_cfg member descriptions.

Member	Description
minor_cnt	Number of Minor Frames in Major Frame.
minor_cfgs	Array of Minor Frame configurations. The length of the array is determined by minor_cnt.

The `gr1553bc_list_cfg` data structure hold the configuration parameters of a descriptor List. The Major and Minor Frames are configured separately. The configuration parameters are used when generating descriptor.

```
struct gr1553bc_list_cfg {
```

```
    unsigned char rt_timeout[31];
    unsigned char bc_timeout;
    int tropt_irq_on_err;
    int tropt_pause_on_err;
    int async_list;
};
```

Table 17.11. gr1553bc_list_cfg member descriptions.

Member	Description
rt_timeout	Number of us timeout tolerance per RT address. The BC has a resolution of 4us.
bc_timeout	Number of us timeout tolerance of broadcast transfers
tropt_irq_on_err	Determines if transfer descriptors should generate IRQ on transfer errors
tropt_pause_on_err	Determines if the list should be paused on transfer error
async_list	Set to non-zero if asynchronous list

17.3.9.2. gr1553bc_list_alloc

Dynamically allocates a List structure (no descriptors) with a maximum number of Major frames supported. The first argument is a pointer to where the newly allocated list pointer will be stored. The second argument determines the maximum number of major frames the List will be able to support.

The list is initialized according to the default configuration.

If the list allocation fails, a negative result will be returned.

17.3.9.3. gr1553bc_list_free

Free a List that has been previously allocated with `gr1553bc_list_alloc()`.

17.3.9.4. gr1553bc_list_config

This function configures List parameters and associate the list with a BC device. The BC device may be used to translate addresses from CPU address to addresses the GR1553B core understand, therefore the list must not be scheduled on another BC device.

Some of the List parameters are used when generating descriptors, as global descriptor parameters. For example all transfer descriptors to a specific RT result in the same time out settings.

The first argument points to a list that is configure. The second argument points to the configuration description, the third argument identifies the BC device that the list will be scheduled on. The layout of the list configuration is described in Table 17.11.

17.3.9.5. gr1553bc_list_link_major

At the end of a Major Frame a unconditional jump to the next Major Frame is inserted by the List API. The List API assumes that a Major Frame should jump to the following Major Frame, however for the last Major Frame the user must tell the API which frame to jump to. The user may also connect Major frames in a more complex way, for example Major Frame 0 and 1 is executed only once so the last Major frame jumps to Major Frame 2.

The Major frame indicated by next will be executed after the Major frame indicated by major. A unconditional jump is inserted to implement the linking.

17.3.9.6. gr1553bc_list_set_major

Major Frames are associated with a number, a Major Frame Number. This function creates an association between a Frame and a Number, all Major Frames must be assigned a number within a List.

The function will link Major[no-1] and Major[no+1] with the Major frame, the linking can be changed by calling `gr1553bc_list_link_major()` after all major frames have been assigned a number.

17.3.9.7. gr1553bc_minor_table_size

This function is used internally by the List API, however it can also be used in an application to calculate the space required by descriptors of a Minor Frame.

The total size of all descriptors in one Minor Frame (in number of bytes) is returned. Descriptors added internally by the List API are also counted.

17.3.9.8. gr1553bc_list_table_size

This function is used internally by the List API, however it can also be used in an application to calculate the total space required by all descriptors of a List.

The total descriptor size of all Major/Minor Frames of the list (in number of bytes) is returned.

17.3.9.9. gr1553bc_list_table_alloc

This function allocates all descriptors needed by a List, either dynamically or by a user provided address. The List is initialized with the new descriptor table, i.e. the software's internal representation is initialized. The descriptors themselves are not initialized.

The second argument *`bdtab_custom`* determines the allocation method. If NULL the API will allocate memory using `malloc()`, if non-zero the value will be taken as the base descriptor address. If bit zero is set the address is assumed to be readable by the GR1553B core, if bit zero is cleared the address is assumed to be readable by the CPU and translated for the GR1553B core. Bit zero makes sense to use on a GR1553B core located on a AMBA-over-PCI bus.

17.3.9.10. gr1553bc_list_table_free

Free previously allocated descriptor table memory.

17.3.9.11. gr1553bc_list_table_build

This function builds all descriptors in a descriptor list. Unused descriptors will be initialized as empty dummy descriptors. Jumps between Minor and Major Frames will be created according to user configuration.

After this call descriptors can be initialized by user.

17.3.9.12. gr1553bc_major_alloc_skel

Allocate a Major Frame and it's Minor Frames according to the configuration pointed to by the second argument.

The pointer to the allocated Major Frame is stored into the location pointed to by the major argument.

The configuration of the Major Frame is determined by the gr1553bc_major_cfg structure, described in Table 17.10.

On success zero is returned, on failure a negative value is returned.

17.3.9.13. gr1553bc_list_freetime

Minor Frames can be configured to handle time slot allocation. This function returns the number of microseconds that is left/unused. The second argument mid determines which Minor Frame.

17.3.9.14. gr1553bc_slot_alloc

Allocate a Slot from a Minor Frame. The Slot location is identified by *`mid`*. If the MID identifies a Minor frame the first free slot is allocated within the minor frame.

The resulting MID of the Slot is stored back to *mid*, the MID can be used in other function call when setting up the Slot. The *mid* argument is thus of in and out type.

The third argument, *timeslot*, determines the time slot that should be allocated to the Slot. If time management is not configured for the Minor Frame a time can still be assigned to the Slot. If the Slot should step to the next Slot directly when finished (no assigned time-slot), the argument must be set to zero. If time management is enabled for the Minor Frame and the requested time-slot is longer than the free time, the call will result in an error (negative result).

The fourth and last argument can optionally be used to get the address of the descriptor used.

17.3.9.15. gr1553bc_slot_free

Return Slot and timeslot allocated from the Minor Frame.

17.3.9.16. gr1553bc_mid_from_bd

Looks up the Slot/Message ID (MID) from a descriptor address. This function may be useful in the interrupt handler, where the address of the descriptor is given.

17.3.9.17. gr1553bc_slot_bd

Looks up descriptor address from MID.

17.3.9.18. gr1553bc_slot_irq_prepare

Prepares a condition descriptor to generate interrupt. Interrupt will not be enabled until `gr1553bc_slot_irq_enable()` is called. The descriptor will be initialized as an unconditional jump to the next descriptor. The Slot can be associated with a custom callback function and an argument. The callback function and argument is stored in the unused fields of the descriptor.

Once enabled and interrupt is generated by the Slot, the callback routine will be called from interrupt context.

The function returns a negative result if failure, otherwise zero is returned.

17.3.9.19. gr1553bc_slot_irq_enable

Enables interrupt of a previously prepared unconditional jump Slot. The Slot is expected to be initialized with `gr1553bc_slot_irq_prepare()`. The descriptor is changed to do a unconditional jump with interrupt.

The function returns a negative result if failure, otherwise zero is returned.

17.3.9.20. gr1553bc_slot_irq_disable

Disable unconditional IRQ point, the descriptor is changed to unconditional JUMP to the following descriptor, without generating interrupt. After disabling the Slot it can be enabled again, or freed.

The function returns a negative result if failure, otherwise zero is returned.

17.3.9.21. gr1553bc_slot_jump

Initialize a Slot with a custom jump condition. The arguments are declared in the table below.

Table 17.12. gr1553bc_list_cfg member descriptions.

Argument	**Description**
list	List that the Slot is located at.
mid	Slot Identification.
condition	Custom condition written to descriptor. See hardware documentation for options.

Argument	Description
to_mid	Slot Identification of the Slot that the descriptor will be jumping to.

Returns zero on success.

17.3.9.22. gr1553bc_slot_exttrig

The BC supports an external trigger signal input which can be used to synchronize 1553 transfers. If used, the external trigger is normally generated by some kind of Time Master. A message slot may be programmed to wait for an external trigger before being executed, this feature allows the user to accurate send time synchronize messages to RTs.

This function initializes a Slot to a dummy transfer with the "Wait for external trigger" bit set.

Returns zero on success.

17.3.9.23. gr1553bc_slot_transfer

Initializes a descriptor to a transfer descriptor. The descriptor is initialized according to the function arguments an the global List configuration parameters. The settings that are controlled on a global level (and not by this function):

- IRQ after transfer error
- IRQ after transfer (not supported, insert separate IRQ slot after this)
- Pause schedule after transfer error
- Pause schedule after transfer (not supported)
- Slot time optional (set when MID allocated), otherwise 0
- (OPTIONAL) Dummy Bit, set using slot_empty() or ..._TT_DUMMY
- RT time out tolerance (managed per RT)

The arguments are declared in the table below.

Table 17.13. gr1553bc_slot_transfer argument descriptions.

Argument	Description
list	List that the Slot is located at
mid	Slot Identification
options	Options: • Retry Mode • Number of retires • Bus selection (A or B) • Dummy bit
tt	Transfer options, see BC transfer type macros in header file: • transfer type • RT src/dst address • RT subaddress • word count • mode code
dptr	Descriptor Data Pointer. Used by Hardware to read or write data to the 1553 bus. If bit zero is set the address is translated by the driver into an address which the hardware can

Argument	Description
	access(may be the case if BC device is located on an AMBA-over-PCI bus), if cleared it is assumed that no translation is required(typical case)

Returns zero on success.

17.3.9.24. gr1553bc_slot_dummy

Manipulate the DUMMY bit of a transfer descriptor. Can be used to enable or disable a transfer descriptor.

The *dummy* argument points to an area used as input and output, as input bit 31 is written to the dummy bit of the descriptor, as output the old value of the descriptor's dummy bit is written.

Returns zero on success.

17.3.9.25. gr1553bc_slot_empty

Create an empty transfer descriptor, with the DUMMY bit set. The time-slot previously allocated is preserved.

Returns zero on success.

17.3.9.26. gr1553bc_slot_update

This function will update a transfer descriptor's status and/or update the data pointer.

If the *dptr* pointer is non-zero the Data Pointer word of the descriptor will be updated with the value of *dptr*. If bit zero is set the driver will translate the data pointer address into an address accessible by the BC hardware. Translation is an option only for AMBA-over-PCI.

If the *stat* pointer is non-zero the Status word of the descriptor will be updated according to the content of *stat*. The old Status will be stored into *stat*. The lower 24-bits of the current Status word may be cleared, and the dummy bit may be set:

```
bd->status = *stat & (bd->status 0xffffff) | (*stat & 0x80000000);
```

Note that the status word is not written (only read) when value pointed to by *stat* is zero.

Returns zero on success.

17.3.9.27. gr1553bc_slot_raw

Custom descriptor initialization. Note that a bad initialization may break the BC driver.

The arguments are declared in the table below.

Table 17.14. gr1553bc_slot_transfer argument descriptions.

Argument	Description
list	List that the Slot is located at
mid	Slot Identification
flags	Determine which words are updated. If bit N is set wordN is written into descriptor wordN, if bit N is zero the descriptor wordN is not modified.
word0	32-bit Word written to descriptor address 0x00
word1	32-bit Word written to descriptor address 0x04
word2	32-bit Word written to descriptor address 0x08
word3	32-bit Word written to descriptor address 0x0C

Returns zero on success.

17.3.9.28. gr1553bc_show_list

Print information about a List to standard out. Each Major Frame's first descriptor for example is printed. This function is used for debugging only.

18. GR1553B Remote Terminal Driver

18.1. Introduction

This section describes the GRLIB GR1553B Remote Terminal (RT) device driver interface. The driver relies on the GR1553B driver and the VxBus framework. The reader is assumed to be well acquainted with MIL-STD-1553 and the GR1553B core. The GR1553RT device driver is included by adding the DRV_GRLIB_GR1553RT component.

18.1.1. GR1553B Remote Terminal Hardware

The GR1553B core supports any combination of the Bus Controller (BC), Bus Monitor (BM) and Remote Terminal (RT) functionality. This driver supports the RT functionality of the hardware, it can be used simultaneously with the Bus Monitor (BM) functionality. When the BM is used together with the RT interrupts are shared between the drivers.

The three functions (BC, BM, RT) are accessed using the same register interface, but through separate registers. In order to shared hardware resources between the three GR1553B drivers, the three depends on a lower level GR1553B driver, see GR1553B driver section.

The driver supports the on-chip AMBA bus and the AMBA-over-PCI bus.

18.2. User Interface

18.2.1. Overview

The RT software driver provides access to the RT core and help with creating memory structures accessed by the RT core. The driver provides the services list below,

- Basic RT functionality (RT address, Bus and RT Status, Enabling core, etc.)
- Event logging support
- Interrupt support (Global Errors, Data Transfers, Mode Code Transfer)
- DMA-Memory configuration
- Sub Address configuration
- Support for Mode Codes
- Transfer Descriptor List Management per RT sub address and transfer type (RX/TX)

The driver sources and definitions are listed in the table below, the path is given relative to the VxWorks source tree `vxworks-6.7/target`.

Table 18.1. RT driver Source location

Filename	Description
src/hwif/grlib/gr1553rt.c	GR1553B RT Driver source
h/hwif/grlib/gr1553rt.h	GR1553B RT Driver interface declaration

18.2.1.1. Accessing an RT device

In order to access an RT core, a specific core must be identified (the driver support multiple devices). The core is opened by calling `gr1553rt_open()`, the open function allocates an RT device by calling the lower level GR1553B driver and initializes the RT by stopping all activity and disabling interrupts. After an RT has been opened it can be configured `gr1553rt_config()`, SA-table configured, descriptor lists assigned to SA, interrupt callbacks registered, and finally communication started by calling `gr1553rt_start()`. Once the RT is started interrupts may be generated, data may be transferred and the event log filled. The communication can be stopped by calling `gr1553rt_stop()`.

When the application no longer needs to access the RT core, the RT is closed by calling `gr1553rt_close()`.

18.2.1.2. Introduction to the RT Memory areas

For the RT there are four different types of memory areas. The access to the areas is much different and involve different latency requirements. The areas are:

- Sub Address (SA) Table
- Buffer Descriptors (BD)
- Data buffers referenced from descriptors (read or written)
- Event (EV) logging buffer

The memory types are described in separate sections below. Generally three of the areas (controlled by the driver) can be dynamically allocated by the driver or assigned to a custom location by the user. Assigning a custom address is typically useful when for example a low-latency memory is required, or the GR1553B core is located on an AMBA-over- PCI bus where memory accesses over the PCI bus will not satisfy the latency requirements by the 1553 bus, instead a memory local to the RT core can be used to shorten the access time. Note that when providing custom addresses the alignment requirement of the GR1553B core must be obeyed, which is different for different areas and sizes. The memory areas are configured using the `gr1553rt_config()` function.

18.2.1.3. Sub Address Table

The RT core provides the user to program different responses per sub address and transfer type through the sub address table (SA-table) located in memory. The RT core consult the SA-table for every 1553 data transfer command on the 1553 bus. The table includes options per sub address and transfer type and a pointer to the next descriptor that let the user control the location of the data buffer used in the transaction. See hardware manual for a complete description.

The SA-table is fixed size to 512 bytes.

Since the RT is required to respond to BC request within a certain time, it is vital that the RT has enough time to lookup user configuration of a transfer, i.e. read SA-table and descriptor and possibly the data buffer as well. The driver provides a way to let the user give a custom address to the sub address table or dynamically allocate it for the user. The default action is to let the driver dynamically allocate the SA-table, the SA-table will then be located in the main memory of the CPU. For RT core's located on an AMBA-over- PCI bus, the default action is not acceptable due to the latency requirement mentioned above.

The SA-table can be configured per SA by calling the `gr1553rt_sa_setopts()` function. The mask argument makes it possible to change individual bit in the SA configuration. This function must be called to enable transfers from/to a sub address. See hardware manual for SA configuration options. Descriptor Lists are assigned to a SA by calling `gr1553rt_list_sa()`.

The indication service can be used to determine the descriptor used in the next transfer, see Section 18.2.1.8.

18.2.1.4. Descriptors

A GR1553B RT descriptor is located in memory and pointed to by the SA-table. The SA-table points out the next descriptor used for a specific sub address and transfer type. The descriptor contains three input fields: Control/Status Word determines options for a specific transfer ans status of a completed transfer; Data buffer pointer, 16-bit aligned; Pointer to next descriptor within sub address and transfer type, or end-of-list marker.

All descriptors are located in the same range of memory, which the driver refers to as the BD memory. The BD memory can by dynamically allocated (located in CPU main memory) by the driver or assigned to a custom location by the user. From the BD memory descriptors for all sub addresses are allocated by the driver. The driver works internally with 16-bit descriptor identifiers allowing 65k descriptor in total. A descriptor is allocated for a specific descriptor List. Each descriptor takes 32 bytes of memory.

The user can build and initialize descriptors using the API function `gr1553rt_bd_init()` and update the descriptor and/or view the status and time of a completed transfer.

Descriptors are managed by a data structure named `gr1553rt_list`. A List is the software representation of a chain of descriptors for a specific sub address and transfer type. Thus, 60 lists in total (two lists per SA, SA0 and SA31 are for mode codes) per RT. The List simplifies the descriptor handling for the user by introducing descriptor numbers (`entry_no`) used when referring to descriptors rather than the descriptor address. Up to 65k descriptors are supported per List by the driver. A descriptor list is assigned to a SA and transfer type by calling `gr1553rt_list_sa()`.

When a List is created and configured a maximal number of descriptors are given, giving the API a possibility to allocate the descriptors from the descriptor memory area configured.

Circular buffers can be created by a chain of descriptors where each descriptor's data buffer is one element in the circular buffer.

18.2.1.5. Data Buffers

Data buffers are not accessed by the driver at all, the address is only written to descriptor upon user request. It is up to the user to provide the driver with valid addresses to data buffers of the required length.

Note that addresses given must be accessible by the hardware. If the RT core is located on a AMBA-over-PCI bus for example, the address of a data buffer from the RT core's point of view is most probably not the same as the address used by the CPU to access the buffer.

18.2.1.6. Event Logging

Transfer events (Transmission, Reception and Mode Codes) may be logged by the RT core into a memory area for (later) processing. The events logged can be controlled by the user at a SA transfer type level and per mode code through the Mode Code Control Register.

The driver API access the eventlog on two occasions, either when the user reads the eventlog buffer using the `gr1553rt_evlog_read()` function or from the interrupt handler, see the interrupt section for more information. The `gr1553rt_evlog_read()` function is called by the user to read the eventlog, it simply copies the current logged entries to a user buffer. The user must empty the driver eventlog in time to avoid entries to be overwritten. A certain descriptor or SA may be logged to help the application implement communication protocols.

The eventlog is typically sized depending the frequency of the log input (logged transfers) and the frequency of the log output (task reading the log). Every logged transfer is described with a 32-bit word, making it quite compact.

The memory of the eventlog does not require as tight latency requirement as the SA-table and descriptors. However the user still is provided the ability to put the eventlog at a custom address, or letting the driver dynamically allocate it. When providing a custom address the start address is given, the area must have room for the configured number of entries and have the hardware required alignment.

Note that the alignment requirement of the eventlog varies depending on the eventlog length.

18.2.1.7. Interrupt service

The RT core can be programmed to interrupt the CPU on certain events, transfers and errors (SA-table and DMA). The driver divides transfers into two different types of events, mode codes and data transfers. The three types of events can be assigned custom callbacks called from the driver's interrupt service routine (ISR), and custom argument can be given. The callbacks are registered per RT device using the functions `gr1553rt_irq_err()`, `gr1553rt_irq_mc()`, `gr1553rt_irq_sa()`. Note that the three different callbacks have different arguments.

Error interrupts are discovered in the ISR by looking at the IRQ event register, they are handled first. After the error interrupt has been handled by the user (user interaction is optional) the RT core is stopped by the driver.

Data transfers and Mode Code transfers are logged in the eventlog. When a transfer-triggered interrupt occurs the ISR starts processing the event log from the first event that caused the IRQ (determined by hardware register) calling the mode code or data transfer callback for each event in the log which has generated an IRQ (determined by the IRQSR bit). Even though both the ISR and the eventlog read function `r1553rt_evlog_read()` processes the eventlog, they are completely separate processes - one does not affect the other. It is up to the user to make sure that events that generated interrupt are not double processed. The callback functions are called in the same order as the event was generated.

Is is possible to configure different callback routines and/or arguments for different sub addresses (1..30) and transfer types (RX/TX). Thus, 60 different callback handlers may be registered for data transfers.

18.2.1.8. Indication service

The indication service is typically used by the user to determine how many descriptors have been processed by the hardware for a certain SA and transfer type. The `gr1553rt_indication()` function returns the next descriptor number which will be used next transfer by the RT core. The indication function takes a sub address and an RT device as input, By remembering which descriptor was processed last the caller can determine how many and which descriptors have been accessed by the BC.

18.2.1.9. Mode Code support

The RT core a number of registers to control and interact with mode code commands. See hardware manual which mode codes are available. Each mode code can be disabled or enabled. Enabled mode codes can be logged and interrupt can be generated upon transmission events. The `gr1553rt_config()` function is used to configure the aforementioned mode code options. Interrupt caused by mode code transmissions can be programmed to call the user through an callback function, see the interrupt Section 18.2.1.7.

The mode codes "Synchronization with data", "Transmit Bit word" and "Transmit Vector word" can be interacted with through a register interface. The register interface can be read with the `gr1553rt_status()` function and selected (or all) bits of the bit word and vector word can be written using `gr1553rt_set_vecword()` function.

Other mode codes can interacted with using the Bus Status Register of the RT core. The register can be read using the `gr1553rt_status()` function and written selectable bit can be written using `gr1553rt_set_bussts()`.

18.2.1.10. RT Time

The RT core has an internal time counter with a configurable time resolution. The finest time resolution of the timer counter is one microsecond. The resolution is configured using the `gr1553rt_config()` function. The current time is read by calling the `gr1553rt_status()` function.

18.2.2. Application Programming Interface

The RT driver API consists of the functions in the table below.

Table 18.2. Data structures

Protoype	Description
`void *gr1553rt_open(int minor)`	Open an RT device by instance number. Returns a handle identifying the specific RT device. The handle is given as input in most functions of the API
`void gr1553rt_close(void *rt)`	Close a previously opened RT device

`int gr1553rt_config(`
`  void *rt,`
`  struct gr1553rt_cfg *cfg)`
	Configure the RT device driver and allocate device memory

`int gr1553rt_start(void *rt)`	Start RT communication, enables Interrupts
`void gr1553rt_stop(void *rt)`	Stop RT communication, disables interrupts
`void gr1553rt_status(`	Get Time, Bus/RT Status and mode code status

Protoype	Description
`void *rt,` `struct gr1553rt_status *status)`	

`int gr1553rt_indication(`
`void *rt,`
`int subadr,`
`int *txeno,`
`int *rxeno)`
	Get the next descriptor that will processed of an RT sub-address and transfer type

`int gr1553rt_evlog_read(`
`void *rt,`
`unsigned int *dst,`
`int max)`
	Copy contents of event log to a user provided data buffer

`void gr1553rt_set_vecword(`
`void *rt,`
`unsigned int mask,`
`unsigned int words)`
	Set all or a selection of bits in the Vector word and Bit word used by the "Transmit Bit word" and "Transmit Vector word" mode codes

`void gr1553rt_set_bussts(`
`void *rt,`
`unsigned int mask,`
`unsigned int sts)`
	Modify a selection of bits in the RT Bus Status register

`void gr1553rt_sa_setopts(`
`void *rt,`
`int subadr,`
`unsigned int mask,`
`unsigned int options)`
	Configures a sub address control word located in the SA-table.

`void gr1553rt_list_sa(`
`struct gr1553rt_list *list,`
`int *subadr,`
`int *tx)`
	Get the Sub address and transfer type of a scheduled list

`void gr1553rt_sa_schedule(`
`void *rt,`
`int subadr,`
`int tx,`
`struct gr1553rt_list *list)`
	Schedule a RX or TX descriptor list on a sub address of a certain transfer type

`int gr1553rt_irq_err(`
`void *rt,`
`gr1553rt_irqerr_t func,`
`void *data)`
	Assign an Error Interrupt handler callback routine and custom argument

`int gr1553rt_irq_mc(`
`void *rt,`
`gr1553rt_irqmc_t func,`
`void *data)`
	Assign a Mode Code Interrupt handler callback routine and custom argument

`int gr1553rt_irq_sa(`
`void *rt,`
`int subadr,`
`int tx,`
`gr1553rt_irq_t func,`
`void *data)`
	Assign a Data Transfer Interrupt handler callback routine and custom argument to a certain sub address and transfer type

`int gr1553rt_list_init(`
`void *rt,`
`struct gr1553rt_list **plist,`
`struct gr1553rt_list_cfg *cfg)`
	Initialize and allocate a descriptor List according to configuration. The List can be used for RX/TX on any sub address.

`int gr1553rt_bd_init(`
`struct gr1553rt_list *list,`
`unsigned short entry_no,`
`unsigned int flags,`
`uint16_t *dptr,`
`unsigned short next)`
	Initialize a Descriptor in a List identified by number.

`int gr1553rt_bd_update(`
`struct gr1553rt_list *list,`
`int entry_no,`
`unsigned int *status,`
`uint16_t **dptr)`
	Update the status and/or the data buffer pointer of a descriptor.

18.2.2.1. Data structures

The `gr1553rt_cfg` data structure is used to configure an RT device. The configuration parameters are described in the table below.

```
struct gr1553rt_cfg {
    unsigned char rtaddress;
    unsigned int modecode;
    unsigned short time_res;
```

```
    void *satab_buffer;
    void *evlog_buffer;
    int evlog_size;
    int bd_count;
    void *bd_buffer;
};
```

Table 18.3. gr1553rt_cfg member descriptions

Member	Description
rtaddress	RT Address on 1553 bus [0..30]
modecode	Mode codes `enable/disable/IRQ/EV-Log`. Each mode code has a 2-bit configuration field. Mode Code Control Register in hardware manual
time_res	Time tag resolution in microseconds
satab_buffer	Sub Address Table (SA-table) allocation setting. Can be dynamically allocated (zero) or custom location (non-zero). If custom location of SA-table is given, the address must be aligned to 10-bit (1kB) boundary and at least 16*32 bytes.
evlog_buffer	Eventlog DMA buffer allocation setting. Can be dynamically allocated (zero) or custom location (non-zero). If custom location of eventlog is given, the address must be of `evlog_size` and aligned to `evlog_size`. See hardware manual.
evlog_size	Length in bytes of Eventlog, must be a multiple of 2. If set to zero event log is disabled, note that enabling logging in SA-table or descriptors will cause failure when eventlog is disabled.
bd_count	Number of descriptors for RT device. All descriptor lists share the descriptors. Maximum is 65K descriptors.
bd_buffer	Descriptor memory area allocation setting. Can be dynamically allocated (zero) or custom location (non-zero). If custom location of descriptors is given, the address must be aligned to 32 bytes and of *`(32 * bd_count)`* bytes size.

The `gr1553rt_list_cfg` data structure hold the configuration parameters of a descriptor List.

```
struct gr1553rt_list_cfg {
    unsigned int bd_cnt;
};
```

Table 18.4. gr1553rt_list_cfg member descriptions

Member	Description
bd_cnt	Number of descriptors in List

The current status of the RT core is stored in the `gr1553rt_status` data structure by the function **gr1553rt_status()**. The fields are described below.

```
struct gr1553rt_status {
    unsigned int status;
    unsigned int bus_status;
    unsigned short synctime;
    unsigned short syncword;
    unsigned short time_res;
    unsigned short time;
};
```

Table 18.5. gr1553rt_status member descriptions

Member	Description
status	Current value of RT Status Register
bus_status	Current value of RT Bus Status Register
synctime	Time Tag when last synchronize with data was received
syncword	Data of last mode code synchronize with data
time_res	Time resolution in microseconds (set by config)

Member	Description
time	Current Time Tag. `(time_res * time)` gives the number of microseconds since last time overflow.

18.2.2.2. gr1553rt_open

Opens a GR1553B RT device identified by instance number, *`minor`*. The instance number is determined by the order in which GR1553B cores with RT functionality are found, the order of the Plug & Play.

A handle is returned identifying the opened RT device, the handle is used internally by the RT driver, it is used as an input parameter rt to all other functions that manipulate the hardware.

Close and Stop an RT device identified by input argument rt previously returned by **gr1553rt_open()**.

18.2.2.3. gr1553rt_close

Close and Stop an RT device identified by input argument rt previously returned by **gr1553rt_open()**.

18.2.2.4. gr1553rt_config

Configure and allocate memory for an RT device. The configuration parameters are stored in the location pointed to by *`cfg`*. The layout of the parameters must follow the `gr1553rt_cfg` data structure, described in Table 18.3.

If memory allocation fails (in case of dynamic memory allocation) the function return a negative result, on success zero is returned.

18.2.2.5. gr1553rt_start

Starts RT communication by enabling the core and enabling interrupts. The user must have configured the driver (RT address, Mode Code, SA-table, lists, descriptors, etc.) before calling this function.

After the RT has been started the configuration function can not be called.

On success this function returns zero, on failure a negative result is returned.

18.2.2.6. gr1553rt_stop

Stops RT communication by disabling the core and disabling interrupts. Further 1553 commands to the RT will be ignored.

18.2.2.7. gr1553rt_status

Read current status of the RT core. The status is written to the location pointed to by status in the format determined by the `gr1553rt_status` structure described in Table 18.5.

18.2.2.8. gr1553rt_indication

Get the next descriptor that will be processed for a specific sub address. The descriptor number is looked up from the descriptor address found the SA-table for the sub address specified by *`subadr`* argument.

The descriptor number of respective transfer type (RX/TX) will be written to the address given by *`txeno`* and/or *`rxeno`*. If end-of-list has been reached, -1 is stored into *`txeno`* or *`rxeno`*.

If the request is successful zero is returned, otherwise a negative number is returned (bad sub address or descriptor).

18.2.2.9. gr1553rt_evlog_read

Copy up to *`max`* number of entries from eventlog into the address specified by *`dst`*. The actual number of entries read is returned. It is important to read out the eventlog entries in time to avoid data loss, the eventlog can be sized so that data loss can be avoided.

Zero is returned when entries are available in the log, negative on failure.

18.2.2.10. gr1553rt_set_vecword

Set a selection of bits in the RT Vector and/or Bit word. The words are used when,

- Vector Word is used in response to "Transmit vector word" BC commands
- Bit Word is used in response to "Transmit bit word" BC commands

The argument *mask* determines which bits are written, and *words* determines the value of the bits written. The lower 16-bits are the `Vector Word`, the higher 16-bits are the Bit Word.

18.2.2.11. gr1553rt_set_bussts

Set a selection of bits of the Bus Status Register. The bits written is determined by the mask bit-mask and the values written is determined by *sts*. Operation:

```
bus_status_reg = (bus_status_reg & ~mask) | (sts & mask)
```

18.2.2.12. gr1553rt_sa_setopts

Configure individual bits of the SA Control Word in the SA-table. One may for example Enable or Disable a SA RX and/or TX. See hardware manual for SA-Table configuration options.

The *mask* argument is a bit-mask, it determines which bits are written and *options* determines the value written.

The *subadr* argument selects which sub address is configured.

Note that SA-table is all zero after configuration, every SA used must be configured using this function.

18.2.2.13. gr1553rt_list_sa

This function looks up the SA and the transfer type of the descriptor list given by `list`. The SA is stored into `subadr`, the transfer type is written into `tx` (TX=1, RX=0).

18.2.2.14. gr1553rt_sa_schedule

This function associates a descriptor list with a sub address (given by *subadr*) and a transfer type (given by *tx*). The first descriptor in the descriptor list is written to the SA-table entry of the SA.

18.2.2.15. gr1553rt_irq_err

his function registers an interrupt callback handler of the Error Interrupt. The handler `func` is called with the argument data when a DMA error or SA-table access error occurs. The callback must follow the prototype of `gr1553rt_irqerr_t` :

```
typedef void (*gr1553rt_irqerr_t)(int err, void *data);
```

Where *err* is the value of the GR1553B IRQ register at the time the error was detected, it can be used to determine what kind of error occurred.

18.2.2.16. gr1553rt_irq_mc

This function registers an interrupt callback handler for Logged Mode Code transmission Interrupts. The handler `func` is called with the argument data when a Mode Code transmission event occurs, note that interrupts must be enabled per Mode Code using **gr1553rt_config()**. The callback must follow the prototype of `gr1553rt_irqmc_t`:

```
typedef void (*gr1553rt_irqmc_t)(
    int mcode,
    unsigned int entry,
    void *data
    );
```

Where *mcode* is the mode code causing the interrupt, entry is the raw event log entry.

18.2.2.17. gr1553rt_irq_sa

Register an interrupt callback handler for data transfer triggered Interrupts, it is possible to assign a unique function and/or data for every SA (given by *subadr*) and transfer type (given by *tx*). The handler *func* is called with the argument *data* when a data transfer interrupt event occurs. Interrupts is configured on a descriptor or SA basis. The callback routine must follow the prototype of gr1553rt_irq_t:

```
typedef void (*gr1553rt_irq_t)(
    struct gr1553rt_list *list,
    unsigned int entry,
    int bd_next,
    void *data
    );
```

Where *list* indicates which descriptor list (Sub Address, transfer type) caused the interrupt event, *entry* is the raw event log entry, *bd_next* is the next descriptor that will be processed by the RT for the next transfer of the same sub address and transfer type.

18.2.2.18. gr1553rt_list_init

Allocate and configure a list structure according to configuration given in *cfg*, see the `gr1553rt_list_cfg` data structure in Table 18.4. Assign the list to an RT device, however not to a sub address yet. The *rt* handle is stored within list.

The resulting descriptor list is written to the location indicated by the *plist* argument.

Note that descriptor are allocated from the RT device, so the RT device itself must be configured using `gr1553rt_config()` before calling this function.

A negative number is returned on failure, on success zero is returned.

18.2.2.19. gr1553rt_bd_init

Initialize a descriptor entry in a list. This is typically done prior to scheduling the list. The descriptor and the next descriptor is given by descriptor indexes relative to the list (*entry_no and next*), see table below for options on *next*. Set bit 30 of the argument *flags* in order to set the IRQEN bit of the descriptor's Control/Status Word. The argument *dptr* is written to the descriptor's Data Buffer Pointer Word.

Note that the data pointer is accessed by the GR1553B core and must therefore be a valid address for the core. This is only an issue if the GR1553B core is located on a AMBA- over-PCI bus, the address may need to be translated from CPU accessible address to hardware accessible address.

Table 18.6. gr1553rt_bd_init next argument description

Values of *next*	**Description**
0xffff	Indicate to hardware that this is the last entry in the list, the next descriptor is set to end-of-list mark (0x3).
0xfffe	Next descriptor (entry_no+1) or 0 is last descriptor in list.
other	The index of the next descriptor.

A negative number is returned on failure, on success a zero is returned.

18.2.2.20. gr1553rt_bd_update

Manipulate and read the Control/Status and Data Pointer words of a descriptor.

If *status* is non-zero, the Control/Status word is swapped with the content pointed to by *status*.

If *dptr* is non-zero, the Data Pointer word is swapped with the content pointed to by *dptr*.

A negative number is returned on failure, on success a zero is returned.

19. GR1553B Bus Monitor Driver

19.1. Introduction

This section describes the GRLIB GR1553B Bus Monitor (BM) device driver interface. The driver relies on the GR1553B driver and the VxBus framework. The reader is assumed to be well acquainted with MIL-STD-1553 and the GR1553B core. The GR1553BM device driver is included by adding the DRV_GRLIB_GR1553BM component.

19.1.1. GR1553B Remote Terminal Hardware

The GR1553B core supports any combination of the Bus Controller (BC), Bus Monitor (BM) and Remote Terminal (RT) functionality. This driver supports the BM functionality of the hardware, it can be used simultaneously with the RT or BC functionality, but not both simultaneously. When the BM is used together with the RT or BC interrupts are shared between the drivers.

The three functions (BC, BM, RT) are accessed using the same register interface, but through separate registers. In order to shared hardware resources between the three GR1553B drivers, the three depends on a lower level GR1553B driver, see GR1553B driver section.

The driver supports the on-chip AMBA bus and the AMBA-over-PCI bus.

19.2. User Interface

19.2.1. Overview

The BM software driver provides access to the BM core and help with accessing the BM log memory buffer. The driver provides the services list below,

- Basic BM functionality (Enabling/Disabling, etc.)
- Filtering options
- Interrupt support (DMA Error, Timer Overflow)
- 1553 Timer handling
- Read BM log

The driver sources and definitions are listed in the table below, the path is given relative to the VxWorks source tree `vxworks-6.7/target`.

Table 19.1. BM driver Source location

Filename	Description
src/hwif/grlib/gr1553bm.c	GR1553B BM Driver source
h/hwif/grlib/gr1553bm.h	GR1553B BM Driver interface declaration

19.2.1.1. Accessing a BM device

In order to access a BM core a specific core must be identified (the driver support multiple devices). The core is opened by calling `gr1553bm_open()`, the open function allocates a BM device by calling the lower level GR1553B driver and initializes the BM by stopping all activity and disabling interrupts. After a BM has been opened it can be configured `gr1553bm_config()` and then started by calling `gr1553bm_start()`. Once the BM is started the log is filled by hardware and interrupts may be generated. The logging can be stopped by calling `gr1553bm_stop()`.

When the application no longer needs to access the BM driver services, the BM is closed by calling `gr1553bm_close()`.

19.2.1.2. BM Log memory

The BM log memory is written by the BM hardware when transfers matching the filters are detected. Each command, Status and Data 16-bit word takes 64-bits of space in the log, into the first 32-bits the current 24-bit 1553 timer is written and to the second 32-bit word status, word type, Bus and the 16-bit data is written. See hardware manual.

The BM log DMA-area can be dynamically allocated by the driver or assigned to a custom location by the user. Assigning a custom address is typically useful when the GR1553B core is located on an AMBA-over-PCI bus where memory accesses over the PCI bus will not satisfy the latency requirements by the 1553 bus, instead a memory local to the BM core can be used to shorten the access time. Note that when providing custom addresses the 8-byte alignment requirement of the GR1553B BM core must be obeyed. The memory areas are configured using the `gr1553bm_config()` function.

19.2.1.3. Accessing the BM Log memory

The BM Log is filled as transfers are detected on the 1553 bus, if the log is not emptied in time the log may overflow and data loss will occur. The BM log can be accessed with the functions listed below.

- gr1553bm_available()
- gr1553bm_read()

A custom handler responsible for copying the BM log can be assigned in the configuration of the driver. The custom routine can be used to optimize the BM log read, for example one may not perhaps not want to copy all entries, search the log for a specific event or compress the log before storing to another location.

19.2.1.4. Time

Th BM core has a 24-bit time counter with a programmable resolution through the `gr1553bm_config()` function. The finest resolution is a microsecond. The BM driver maintains a 64-bit 1553 time. The time can be used by an application that needs to be able to log for a long time. The driver must detect every overflow in order maintain the correct 64-bit time, the driver gives users two different approaches. Either the timer overflow interrupt is used or the user must guarantee to call the `gr1553bm_time()` function at least once before the second time overflow happens. The timer overflow interrupt can be enabled from the `gr1553bm_config()` function.

The current 64-bit time can be read by calling `gr1553bm_time()`.

The application can determine the 64-bit time of every log entry by emptying the complete log at least once per timer overflow.

19.2.1.5. Filtering

The BM core has support for filtering 1553 transfers. The filter options can be controlled by fields in the configuration structure given to `gr1553bm_config()`.

19.2.1.6. Interrupt service

The BM core can interrupt the CPU on DMA errors and on Timer overflow. The DMA error is unmasked by the driver and the Timer overflow interrupt is configurable. For the DMA error interrupt a custom handler may be installed through the `gr1553bm_config()` function. On DMA error the BM logging will automatically be stopped by a call to `gr1553bm_stop()` from within the ISR of the driver.

19.2.2. Application Programming Interface

The BM driver API consists of the functions in the table below.

Table 19.2. function prototypes

Prototype	Description
`void *gr1553bm_open(int minor)`	Open a BM device by instance number. Returns a handle identifying the specific BM device opened. The handle is given as input parameter *bm* in all other functions of the API
`void gr1553bm_close(void *bm)`	Close a previously opened BM device

`int gr1553bm_config(`
`  void *bm,`
`  struct gr1553bm_cfg *cfg)`
	Configure the BM device driver and allocate BM log DMA-memory

`int gr1553bm_start(void *bm)`	Start BM logging, enables Interrupts
`void gr1553bm_stop(void *bm)`	Stop BM logging, disables interrupts

`void gr1553bm_time(`
`  void *bm,`
`  uint64_t *time)`
	Get 1553 64-bit Time maintained by the driver. The lowest 24-bits are taken directly from the BM timer register, the most significant 40-bits are taken from a software counter.

`int gr1553bm_available(`
`  void *bm,`
`  int *nentries)`
	The current number of entries in the log is stored into `nentries`.

`int gr1553bm_read(`
`  void *bm,`
`  struct gr1553bm_entry *dst,`
`  int *max)`
	Copy contents a maximum number (*max*) of entries from the BM log to a user provided data buffer (dst). The actual number of entries copied is stored into *max*.

19.2.2.1. Data structures

The `gr1553bm_cfg` data structure is used to configure the BM device and driver. The configuration parameters are described in the table below.

```
struct gr1553bm_config {
    uint8_t time_resolution;
    int time_ovf_irq;
    unsigned int filt_error_options;
    unsigned int filt_rtadr;
    unsigned int filt_subadr;
    unsigned int filt_mc;
    unsigned int buffer_size;
    void *buffer_custom;
    bmcopy_func_t copy_func;
    void *copy_func_arg;
    bmisr_func_t dma_error_isr;
    void *dma_error_arg;
};
```

Table 19.3. gr1553bm_config member descriptions.

Member	Description
time_resolution	8-bit time resolution, the BM will update the time according to this setting. 0 will make the time tag be of highest resolution (no division), 1 will make the BM increment the time tag once for two time ticks (div with 2), etc.
time_ovf_irq	Enable Time Overflow IRQ handling. Setting this to 1 makes the driver to update the 64-bit time by it self, it will use time overflow IRQ to detect when the 64-bit time counter must be incremented. If set to zero, the driver expect the user to call gr1553bm_time() regularly, it must be called more often than the time overflows to avoid an incorrect time.
filt_error_options	Bus error log options: bit0,4-31 = reserved, set to zero Bit1 = Enables logging of Invalid mode code errors Bit2 = Enables logging of Unexpected Data errors Bit3 = Enables logging of Manchester/parityerrors
filt_rtadr	RT Subaddress filtering bit mask, bit definition: 31: Enables logging of mode commands on subadr 31 1..30: BitN enables/disables logging of RT subadr N 0: Enables logging of mode commands on subadr 0

Member	Description
filt_mc	Mode code Filter, is written into "BM RT Mode code filter" register, please see hardware manual for bit declarations.
buffer_size	Size of buffer in bytes, must be aligned to 8-byte boundary.
buffer_custom	Custom BM log buffer location, must be aligned to 8-byte and be of buffer_size length. If NULL dynamic memory allocation is used.
copy_func	Custom Copy function, may be used to implement a more effective/ custom way of copying the DMA buffer. For example the DMA log may need to processed at the same time when copying.
copy_func_arg	Optional Custom Data passed onto `copy_func()`
dma_error_isr	Custom DMA error function, note that this function is called from Interrupt Context. Set to NULL to disable this callback.
dma_error_arg	COptional Custom Data passed on to `dma_error_isr()`

```
struct gr1553bm_entry {
    uint32_t time;
    uint32_t data;
};
```

Table 19.4. gr1553bm_entry member descriptions.

Member	Description
time	Time of word transfer entry. Bit31=1, bit 30..24=0, bit 23..0=time
data	Transfer status and data word

Bits	Description
31	Zero
30..20	Zero
19	0=BusA, 1=BusB
18..17	Word Status: 00=Ok, 01=Manchester error, 10=Parity error
16	Word type: 0=Data, 1=Command/ Status
15..0	16-bit Data on detected on bus

19.2.2.2. gr1553bm_open

Opens a GR1553B BM device identified by instance number, `minor`. The instance number is determined by the order in which GR1553B cores with BM functionality are found, the order of the Plug & Play.

A handle is returned identifying the opened BM device, the handle is used internally by the driver, it is used as an input parameter *`bm`* to all other functions that manipulate the hardware.

This function initializes the BM hardware to a stopped/disable level.

19.2.2.3. gr1553bm_close

Close and Stop a BM device identified by input argument bm previously returned by `gr1553bm_open()`.

19.2.2.4. gr1553bm_config

Configure and allocate the log DMA-memory for a BM device. The configuration parameters are stored in the location pointed to by *`cfg`*. The layout of the parameters must follow the `gr1553bm_config` data structure, described in Table 19.3.

If BM device is started or memory allocation fails (in case of dynamic memory allocation) the function return a negative result, on success zero is returned.

19.2.2.5. gr1553bm_start

Starts 1553 logging by enabling the core and enabling interrupts. The user must have configured the driver (log buffer, timer, filtering, etc.) before calling this function.

After the BM has been started the configuration function can not be called.

On success this function returns zero, on failure a negative result is returned.

19.2.2.6. gr1553bm_stop

Stops 1553 logging by disabling the core and disabling interrupts. Further 1553 transfers will be ignored.

19.2.2.7. gr1553bm_time

This function reads the driver's internal 64-bit 1553 Time. The low 24-bit time is acquired from BM hardware, the MSB is taken from a software counter internal to the driver. The counter is incremented every time the Time overflows by:

- using "Time overflow" IRQ if enabled in user configuration
- by checking "Time overflow" IRQ flag (IRQ is disabled), it is required that user calls this function before the next timer overflow. The software can not distinguish between one or two timer overflows. This function will check the overflow flag and increment the driver internal time if overflow has occurred since last call.

This function update software time counters and store the current time into the address indicated by the argument time.

19.2.2.8. gr1553bm_available

Copy up to *max* number of entries from BM log into the address specified by *dst*. The actual number of entries read is returned in the location of *max* (zero when no entries available). *The max* argument is thus in/out. It is important to read out the log entries in time to avoid data loss, the log can be sized so that data loss can be avoided.

Zero is returned on success, on failure a negative number is returned.

19.2.2.9. gr1553bm_read

Copy up to max number of entries from BM log into the address specified by dst. The actual number of entries read is returned in the location of max (zero when no entries available). The max argument is thus in/out. It is important to read out the log entries in time to avoid data loss, the log can be sized so that data loss can be avoided.

Zero is returned on success, on failure a negative number is returned.

20. PCI Support

20.1. Overview

This section gives an introduction to the available PCI Host drivers and board drivers available in the SPARC/VxWorks 6.7 distribution. Host drivers are used to access the PCI bus, and PCI board drivers are used to access one particular board on the PCI bus.

20.2. Source code

Sources are located at `vxworks-6.7/target/src/hwif`. More specifically the LEON VxBus related sources are located as indicated by the table below, all paths are given relative to `vxworks-6.7/target`.

Table 20.1. LEON specific PCI sources

Location	Description
src/drv/pci	VxWorks PCI Library and PCI auto configuration library.
src/hwif/vxbus/vxbPci.c	PCI bus controller layer, used to implement a PCI Host driver.
src/hwif/busCtlr/grlibGrpci.c	GRPCI PCI Host bus controller driver.
src/hwif/busCtlr/grlibPcif.c	GRLIB PCIF (ACTEL PCIF Core wrapper) PCI host bus controller driver.
src/hwif/busCtrl/at697pci.c	AT697 PCI host bus controller driver.
src/hwif/pcitargets/ gr_rasta_io.c	GR-RASTA-IO PCI Board driver.
src/hwif/pcitargets/ gr_rasta_adcdac.c	GR-RASTA-ADCDAC PCI Board driver.
src/hwif/pcitargets/gr_701.c	GR-701 Companion chip PCI Board driver.

20.3. PCI Host drivers

20.3.1. Overview

This section describes PCI Host support in VxWorks 6.7 for SPARC/LEON processors. The supported PCI Host core are summarized in the table below. All PCI Host drivers support both MMU and non-MMU BSPs where available. Note that only one PCI Host driver can be included in a project at the same time.

Table 20.2. SPARC/LEON PCI support

PCI Driver	non-MMU	MMU	Component
GRLIB GRPCI	Yes	Yes	INCLUDE_GRLIB_GRPCI_PCI
ACTEL CorePCIF PCIF	Yes	Yes	INCLUDE_GRLIB_PCIF_PCI
AT697 PCI	Yes	No	INCLUDE_AT697_PCI

All drivers implement an interface towards the VxWorks PCI library. The interface to the user is therefore similar, however there exists differences in hardware, what the cores supports, for example DMA and multiple bus (bridge) support may differ. The PCI auto configuration library is used default to set up PCI resources.

Note that functions for PCI DMA are not provided at the point of writing.

Note that only one PCI Host driver can be included in a project.

20.3.2. PCI configuration

The PCI support is included by adding the INCLUDE_PCI component from the workbench kernel configuration "Bus Support" section under hardware. One must also select one of the three Host PCI

drivers available. The PCI address space can be configured from `config.h` or from the workbench kernel configuration utility INCLUDE_PCI_PARAMS component. VxWorks separates the PCI addresses by type, I/O space or memory space or non-prefetchable memory space. How the address space should be set up differ between systems depending on the needs of the PCI boards connected to the bus. The regions may not overlap.

The AMBA address space used to access the PCI address space is mapped 1:1.

The default is to map the SRAM or SDRAM of the CPU board to PCI address space on a 1:1 basis. This is to make the main memory available for cores on the PCI bus that supports DMA transactions.

On MMU systems the AMBA address window used to access PCI space has the same virtual address as the physical, thus virtual addresses is mapped 1:1 to physical addresses. This removes the need for device drivers to map their hardware to virtual space since all of PCI space is already mapped by the AMBA routines during start up.

Interrupts are either, shared where multiple PCI interrupts (INT#A..D) share the same system interrupt, or non-shared where every used interrupt is assigned one unique system IRQ. The non-shared systems use GPIO interrupt pins to connect one PCI interrupt to one system interrupt. PCI interrupts may also be shared between boards, For example two PCI board may both trigger INTB#, this is possible due to the level sensitivity of PCI interrupts.

For systems where the PCI Host core take in the PCI interrupt the USE_PCI_SHARED_IRQ parameter should be set to FALSE. On systems where GPIO is used to take one or more PCI interrupts the USE_PCI_SHARED_IRQ should be set to TRUE and the PCI_INTX_IRQ must reflect the system IRQ that the GPIO core will trigger when PCI INTX# interrupt is caused. The PCI_INT[N]_IRQ number represents the system IRQ the PCI_INT[N]# is connected to, all the system IRQs given will be enabled and configured as low level sensitive. Unconnected PCI IRQs must be set to 0xff.

20.3.3. MMU Considerations

Since the BSP AMBA initialization routines maps AHB slave I/O spaces but not AHB slave memory spaces, the configuration space is already mapped into virtual address space. However, the PCI Window is a memory space area and therefore not mapped, this requires the PCI Host drivers to map in the desired areas. After the PCI Library has initialized and allocated resources to all PCI boards the PCIF and GRPCI host drivers map each enabled PCI board's BARs into the virtual address space of the CPU using the MMU. The mapping is done 1:1. This way the CPU can always access the configured PCI memory areas, an MMU trap will occur if accesses are made outside of the BARs.

The AMBA address window used to access PCI space has the same virtual address as the physical, thus virtual addresses is mapped 1:1 to physical addresses. This removes the need for device drivers to map their hardware to virtual space.

20.3.4. GRPCI Host driver

The GRPCI core has an AMBA address window that translates into PCI space accesses. The PCI board are accesses somewhere in the space depending on how the PCI Library has allocated the memory resources, the BARs. After resource allocation has finished the GRPCI driver maps each BAR 1:1 into virtual address space in a MMU system. After the mapping has been done, the non-MMU and MMU BSP drivers may access the PCI BARs the same way.

The byte twisting options may be used to access a little endian PCI bus without doing byte twisting in the CPU. SPARC is big endian and normally byte twisting is required.

If the GRPCI shared interrupt service is used, the GRPCI is responsible to route PCI interrupt to a system interrupt and the PCI_INT#_IRQ parameters will be ignored and the IRQ number will be taken from the AMBA Plug&Play GRPCI IRQ number. This is the default behavior and GRPCI_VXBUS_GPIO_INTERRUPT must be set to FALSE.

However, if the GRPCI is not connected to the PCI interrupts, but GPIO is used to connect system interrupts with PCI interrupts, one must set the PCI_INT#_IRQ to match the hardware design. The PCI_INT[N]_IRQ

number represents the system IRQ the PCI_INT[N]# is connected to, all the system IRQs given will be enabled and configured as low level sensitive. Unconnected PCI IRQs are set to 0xff.

Note that in a GRLIB system the GPIO number is the same as the IRQ number in a system.

20.3.5. PCIF Host driver

The GRLIB PCIF core is an AMBA wrapper for ACTEL CorePCIF. The interface the PCIF wrapper is similar to the GRPCI interface. The PCIF does not support changing byte twisting and PCI bridges. See GRPCI section above for guidelines.

20.3.6. AT697 PCI Host driver interface

The LEON2 AT697 PCI has a fixed AMBA address window 0xA0000000-0xF0000000 translated into PCI addresses without address modifications. The address regions and sizes available to VxWorks must be configured using the INCLUDE_PCI_PARAMS component described above.

Interrupt are generated using the GPIO interface which is connected to one of the I/O interrupt sources. The GPIO number used can be changed by setting PCI_SHARED_IRQ_PIO, the system interrupt connected to the GPIO interrupt can be changed by setting PCI_SHARED_IRQ.

Note that the BARs used to access the AT697 from PCI space is limited to 16Mb, this may be a problem when the AT697 main memory is larger than 16Mb. It is because drivers may set up that PCI masters should do DMA to the AT697 main memory, however the drivers may not know that there is a 16MB limit since they use the standard malloc() routine. The Aeroflex Gaisler drivers that support DMA capable cores often provide means for the user to provide a custom address that can be used to solve this issue.

20.4. PCI Board drivers

20.4.1. Overview

This section describes the Aeroflex Gaisler PCI board support in VxWorks 6.7. The board drivers are not dependent of the PCI Host driver, as long as it is a VxBus host driver.

All drivers in this section targets boards that are based on GRLIB. The drivers set up the target system so that interrupts and access to memory and GRLIB cores is possible, an AMBA Plug&Play VxBus bus controller driver is implemented upon the generic AMBA Plug&Play layer described in Chapter 8.

20.4.2. GR-RASTA-IO PCI driver

The driver implements a VxBus AMBA Plug&Play bus controller, supporting interrupt handling, address translation, bus frequency, driver resources and so on. All standard on-chip VxBus drivers (CAN, SpaceWire, ..) is reused. The devices are registered to a different path than the normal on-chip core path, all paths are prefixed with /gr_rasta_io/NUM, where NUM identifies a specific GR-RASTA-IO board when multiple GR-RASTA-IO boards are used in the same system. Thus, opening the second GRSPW device on the first GR-RASTA-IO board the path `/gr_rasta_io/0/grspw/1` must be used.

The driver is included by adding the INCLUDE_PCIDEV_GR_RASTA_IO component, other configuration parameters are similar to the standard on-chip driver's component configuration parameters, please see their respective documentation.

Note that including Timer support may result in that the timers are used by the standard Timer VxWorks services (sysClk, sysTimestamp, sysAuxClk...).

20.4.2.1. Show routine

The driver has a show routine showing the PCI BARs, AMBA Plug&Play bus information and current IRQ assignment of all found GR-RASTA-IO boards. The show routine can for example be called directly from the VxWorks C shell:

```
-> grRastaIoShow()
```

20.4.3. GR-RASTA-ADCDAC PCI driver

The driver implements a VxBus AMBA Plug&Play bus controller, supporting interrupt handling, address translation, bus frequency, driver resources and so on. All standard on-chip VxBus drivers (ADC/ADC, CAN, ..) is reused. The devices are registered to a different path than the normal on-chip core path, all paths are prefixed with /gr_rasta_adcdac/NUM, where NUM identifies a specific GR-RASTA-ADCDAC board when multiple GR-RASTA-ADCDAC boards are used in the same system. Thus, opening the first GRCAN device on the second GR-RASTA-ADCDAC board the path `/gr_rasta_adcdac/1/grcan/0` must be used.

The driver is included by adding the INCLUDE_PCIDEV_GR_RASTA_ADCDAC component, other configuration parameters are similar to the standard on-chip driver's component configuration parameters, please see their respective documentation.

Note that including Timer support may result in that the timers are used by the standard Timer VxWorks services (sysClk, sysTimestamp, sysAuxClk...).

20.4.3.1. Show routine

The driver has a show routine showing the PCI BARs, AMBA Plug&Play bus information and current IRQ assignment of all found GR-RASTA-ADCDAC boards. The show routine can for example be called directly from the VxWorks C shell:

```
-> grRastaAdcdacShow()
```

20.4.4. GR-701 PCI driver

The driver implements a VxBus AMBA Plug&Play bus controller, supporting interrupt handling, address translation, bus frequency, driver resources and so on. All standard on-chip VxBus drivers (CAN, SpaceWire, ..) is reused. The devices are registered to a different path than the normal on-chip core path, all paths are prefixed with /gr701/NUM, where NUM identifies a specific GR-701 board when multiple GR-701 boards are used in the same system. Thus, opening the second GRSPW device on the first GR-RASTA-IO board the path `/gr701/0/grspw/1` must be used.

The driver is included by adding the INCLUDE_PCIDEV_GR701 component, other configuration parameters are similar to the standard on-chip driver's component configuration parameters, please see their respective documentation.

Note that including Timer support may result in that the timers are used by the standard Timer VxWorks services (sysClk, sysTimestamp, sysAuxClk...).

20.4.4.1. Show routine

The driver has a show routine showing the PCI BARs, AMBA Plug&Play bus information and current IRQ assignment of all found GR-701 boards. The show routine can for example be called directly from the VxWorks C shell:

```
-> gr701Show()
```

21. Support

For Support, contact the Aeroflex Gaisler support team at support@gaisler.com.

22. Disclaimer

Aeroflex Gaisler AB, reserves the right to make changes to any products and services described herein at any time without notice. Consult Aeroflex or an authorized sales representative to verify that the information in this document is current before using this product. Aeroflex does not assume any responsibility or liability arising out of the application or use of any product or service described herein, except as expressly agreed to in writing by Aeroflex; nor does the purchase, lease, or use of a product or service from Aeroflex convey a license under any patent rights, copyrights, trademark rights, or any other of the intellectual rights of Aeroflex or of third parties.